# logo-sql08.gif

Analysis Services Performance Guide

SQL Server Technical Article

**Writers:** Richard Tkachuk and Thomas Kejser

**Contributors and Technical Reviewers:**

T.K. Anand

Marius Dumitru

Greg Galloway

Siva Harinath

Denny Lee

Edward Melomed

Akshai Mirchandani

Mosha Pasumansky

Carl Rabeler

Elizabeth Vitt

Sedat Yogurtcuoglu

Anne Zorner

**Published:** October 2008

**Applies to:** SQL Server 2008

**Summary:** This white paper describes how application developers can apply query and processing performance-tuning techniques to their SQL Server 2008 Analysis Services OLAP solutions.

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, Excel, SQL Server, Visual Basic, Windows, and Windows Server are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

1 Introduction

2 Understanding the Query Processor Architecture

2.1 Session Management

2.2 Job Architecture

2.3 Query Processor

2.3.1 Query Processor Cache

2.3.2 Query Processor Internals

2.4 Data Retrieval

3 Enhancing Query Performance

3.1 Baselining Query Speeds

3.2 Diagnosing Query Performance Issues

3.3 Optimizing Dimensions

3.3.1 Identifying Attribute Relationships

3.3.2 Using Hierarchies Effectively

3.4 Maximizing the Value of Aggregations

3.4.1 Detecting Aggregation Hits

3.4.2 How to Interpret Aggregations

3.4.3 Building Aggregations

3.5 Using Partitions to Enhance Query Performance

3.5.1 Introduction

3.5.2 Partition Slicing

3.5.3 Aggregation Considerations for Multiple Partitions

3.5.4 Distinct Count Partition Design

3.5.5 Partition Sizing

3.6 Optimizing MDX

3.6.1 Diagnosing the Problem

3.6.2 Calculation Best Practices

3.7 Cache Warming

3.8 Improving Multiple-User Performance

3.8.1 Increasing Query Parallelism

3.8.2 Memory Heap Type

3.8.3 Blocking Long-Running Queries

3.8.4 Network Load Balancing and Read-Only Databases

4 Understanding and Measuring Processing

4.1 Processing Job Overview

4.2 Baselining Processing

4.2.1 Performance Monitor Trace

4.2.2 Profiler Trace

4.3 Determining Where You Spend Processing Time

5 Enhancing Dimension Processing Performance

5.1 Understanding Dimension Processing Architecture

5.2 Dimension-Processing Commands

5.3 Dimension Processing Tuning Flow Chart

5.4 Dimension Processing Performance Best Practices

5.4.1 Use SQL Views to Implement Query Binding for Dimensions

5.4.2 Optimize Attribute Processing Across Multiple Data Sources

5.4.3 Reduce Attribute Overhead

5.4.4 Use the KeyColumns, ValueColumn, and NameColumn Properties Effectively

5.4.5 Remove Bitmap Indexes

5.4.6 Turn Off the Attribute Hierarchy and Use Member Properties

5.5 Tuning the Relational Dimension Processing Query

6 Enhancing Partition Processing Performance

6.1 Understanding the Partition Processing Architecture

6.2 Partition-Processing Commands

6.3 Partition Processing Tuning Flow Chart

6.4 Partition Processing Performance Best Practices

6.4.1 Optimizing Data Inserts, Updates, and Deletes

6.4.2 Picking Efficient Data Types in Fact Tables

6.5 Tuning the Relational Partition Processing Query

6.5.1 Getting Rid of Joins

6.5.2 Getting Relational Partitioning Right

6.5.3 Getting Relational Indexing Right

6.5.4 Using Index FILLFACTOR = 100 and Data Compression

6.6 Eliminating Database Locking Overhead

6.7 Optimizing Network Throughput

6.8 Improving the I/O Subsystem

6.9 Increasing Concurrency by Adding More Partitions

6.10 Adjusting Maximum Number of Connections

6.11 Adjusting ThreadPool and CoordinatorExecutionMode

6.12 Adjusting BufferMemoryLimit

6.13 Tuning the Process Index Phase

6.13.1 Avoid Spilling Temporary Data to Disk

6.13.2 Eliminate I/O Bottlenecks

6.13.3 Add Partitions to Increase Parallelism

6.13.4 Tune Threads and AggregationMemorySettings

7 Tuning Server Resources

7.1 Using PreAllocate

7.2 Disable Flight Recorder

7.3 Monitoring and Adjusting Server Memory

8 Conclusion

# Introduction

Since Microsoft® SQL Server® Analysis Services query and processing performance tuning is a fairly broad subject, this white paper organizes performance tuning techniques into the following three segments.

[**Enhancing Query Performance**](#_4d34og8) - Query performance directly impacts the quality of the end user experience. As such, it is the primary benchmark used to evaluate the success of an online analytical processing (OLAP) implementation. Analysis Services provides a variety of mechanisms to accelerate query performance, including aggregations, caching, and indexed data retrieval. In addition, you can improve query performance by optimizing the design of your dimension attributes, cubes, and Multidimensional Expressions (MDX) queries.

**Enhancing Processing Performance** - Processing is the operation that refreshes data in an Analysis Services database. The faster the processing performance, the sooner users can access refreshed data. Analysis Services provides a variety of mechanisms that you can use to influence processing performance, including efficient dimension design, effective aggregations, partitions, and an economical processing strategy (for example, incremental vs. full refresh vs. proactive caching).

**Tuning Server Resources** – There are several engine settings that can be tuned that affect both querying and processing performance.

# Understanding the Query Processor Architecture

To make the querying experience as fast as possible for end users, the Analysis Services querying architecture provides several components that work together to efficiently retrieve and evaluate data. Figure 1 identifies the three major operations that occur during querying: session management, MDX query execution, and data retrieval, as well as the server components that participate in each operation.

**Figure 1 Analysis Services query processor architecture**

## Session Management

Client applications communicate with Analysis Services using XML for Analysis (XMLA) over TCP/IP or HTTP. Analysis Services provides an XMLA listener component that handles all XMLA communications between Analysis Services and its clients. The Analysis Services Session Manager controls how clients connect to an Analysis Services instance. Users authenticated by the Windows® operating system and who have access to at least one database can connect to Analysis Services. After a user connects to Analysis Services, the Security Manager determines user permissions based on the combination of Analysis Services roles that apply to the user. Depending on the client application architecture and the security privileges of the connection, the client creates a session when the application starts, and then reuses the session for all of the user’s requests. The session provides the context under which client queries are executed by the query processor. A session exists until it is closed by the client application or the server.

## Job Architecture

Analysis Services uses a centralized job architecture to implement querying and processing operations. A *job* is a generic unit of processing or querying work. A job can have multiple levels of nested child jobs depending on the complexity of the request.

During processing operations, for example, a job is created for the object that you are processing, such as a dimension. A dimension job can then spawn several child jobs that process the attributes in the dimension. During querying, jobs are used to retrieve fact data and aggregations from the partition to satisfy query requests. For example, if you have a query that accesses multiple partitions, a parent or coordinator job is generated for the query itself along with one or more child jobs per partition.

**Figure 2 Job architecture**

Generally speaking, executing more jobs in parallel has a positive impact on performance as long as you have enough processor resources to effectively handle the concurrent operations as well as sufficient memory and disk resources. The maximum number of jobs that can execute in parallel for the current operation operations (including both processing and querying) is determined by the **CoordinatorExecutionMode** property:

* A negative specifies the maximum number of parallel jobs that can start per core per operation.
* A value of zero indicates no limit.
* A positive value specifies an absolute number of parallel jobs that can start per server.

The default value for the **CoordinatorExecutionMode** is -4, which indicates that four jobs will be started in parallel per core. This value is sufficient for most server environments. If you want to increase the level of parallelism in your server, you can increase the value of this property either by increasing the number of jobs per processor or by setting the property to an absolute value.

While this globally increases the number of jobs that can execute in parallel, **CoordinatorExecutionMode** is not the only property that influences parallel operations. You must also consider the impact of other global settings such as the **MaxThreads** server properties that determine the maximum number of querying or processing threads that can execute in parallel (see Improving Multiple-User Performance for more information about thread settings). In addition, at a more granular level, for a given processing operation, you can specify the maximum number of processing tasks that can execute in parallel using the **MaxParallel** command. These settings are discussed in more detail in the sections that follow.

## Query Processor

The query processor executes MDX queries and generates a cellset or rowset in return. This section provides an overview of how the query processor executes queries. For more information about optimizing MDX, see Optimizing MDX.

To retrieve the data requested by a query, the query processor builds an execution plan to generate the requested results from the cube data and calculations. There are two major different types of query execution plans, and which one is chosen by the engine can have a significant impact on performance. For more information, see Subspace Computation.

To communicate with the storage engine, the query processor uses the execution plan to translate the data request into one or more subcube requests that the storage engine can understand. A subcube is a logical unit of querying, caching, and data retrieval – it is a subset of cube data defined by the crossjoin of one or more members from a single level of each attribute hierarchy. One or more members from a single level are also sometimes called a *single grain* or *single granularity*. An MDX query can be resolved into multiple subcube requests depending the attribute granularities involved and calculation complexity; for example, a query involving every member of the Country attribute hierarchy (assuming it’s not a parent child hierarchy) would be split into two subcube requests: one for the All member and another for the countries.

As the query processor evaluates cells, it uses the query processor cache to store calculation results. The primary benefits of the cache are to optimize the evaluation of calculations and to support the reusage of calculation results across users (with the same security roles). To optimize cache reusage, the query processor manages three cache layers that determine the level of cache reusability: global, session, and query.

### Query Processor Cache

During the execution of an MDX query, the query processor stores calculation results in the query processor cache. The primary benefits of the cache are to optimize the evaluation of calculations and to support reuse of calculation results across users. To understand how the query processor uses caching during query execution, consider the following example. You have a calculated member called Profit Margin. When an MDX query requests Profit Margin by Sales Territory, the query processor caches the nonnull Profit Margin values for each Sales Territory. To manage the reuse of the cached results across users, the query processor distinguishes different contexts in the cache:

* **Query Context**—contains the result of any calculations created by using the WITH keyword within a query. The query context is created on demand and terminates when the query is over. Therefore, the cache of the query context is not shared across queries in a session.
* **Session Context** —contains the result of any calculations created by using the CREATE statement within a given session. The cache of the session context is reused from request to request in the same session, but it is not shared across sessions.
* **Global Context** —contains the result of any calculations that are shared among users. The cache of the global context can be shared across sessions if the sessions share the same security roles.

The contexts are tiered in terms of their level of reusage. At the top, the query context is can be reused only within the query. At the bottom, the global context has the greatest potential for reusage across multiple sessions and users.

**Figure 3 Cache context layers**

During execution, every MDX query must reference all three contexts to identify all of the potential calculations and security conditions that can impact the evaluation of the query. For example, to resolve a query that contains a query calculated member, the query processor creates a query context to resolve the query calculated member, creates a session context to evaluate session calculations, and creates a global context to evaluate the MDX script and retrieve the security permissions of the user who submitted the query. Note that these contexts are created only if they aren’t already built. After they are built, they are reused where possible.

Even though a query references all three contexts, it can only use the cache of a single context. This means that on a per-query basis, the query processor must select which cache to use. The query processor always attempts to use the broadly applicable cache depending on whether or not it detects the presence of calculations at a narrower context.

If the query processor encounters calculations created at query time, it always uses the query context, even if a query also references calculations from the global context (there is an exception to this – queries with query calculated members of the form Aggregate(<set>) do share the session cache). If there are no query calculations, but there are session calculations, the query processor uses the session cache. The query processor selects the cache based on the presence of any calculation in the scope. This behavior is especially relevant to users with MDX-generating front-end tools. If the front-end tool creates any session calculations or query calculations, the global cache is not used, even if you do not specifically use the session or query calculations.

There are other calculation scenarios that impact how the query processor caches calculations. When you call a stored procedure from an MDX calculation, the engine always uses the query cache. This is because stored procedures are nondeterministic (meaning that there is no guarantee what the stored procedure will return). As a result, nothing will be cached globally or in the session cache. Rather, the calculations will be stored in the query cache. In addition, the following scenarios determine how the query processor caches calculation results:

* Use of cell security, any of the **UserName**, **StToSet**, or **LookupCube** functions in the MDX script or in the dimension or cell security definition disable the global cache (this means that just one expression using these functions disables global caching for the entire cube).
* If visual totals are enabled for the session by setting the default MDX Visual Mode property in the Analysis Services connection string to 1, the query processor uses the query cache for all queries issued in that session.
* If you enable visual totals for a query by using the MDX **VisualTotals** function, the query processor uses the query cache.
* Queries that use the subselect syntax (SELECT FROM SELECT) or are based on a session subcube (CREATE SUBCUBE) result in the query or, respectively, session cache to be used.
* Arbitrary shapes can only use the query cache if they are used in a subselect, in the WHERE clause, or in a calculated member. An arbitrary shape is any set that cannot be expressed as a crossjoin of members from the same level of an attribute hierarchy. For example, {(Food, USA), (Drink, Canada)} is an arbitrary set, as is {customer.geography.USA, customer.geography.[British Columbia]}. Note that an arbitrary shape on the query axis does not limit the use of any cache.

Based on this behavior, when your querying workload can benefit from reusing data across users, it is a good practice to define calculations in the global scope. An example of this scenario is a structured reporting workload where you have few security roles. By contrast, if you have a workload that requires individual data sets for each user, such as in an HR cube where you have many security roles or you are using dynamic security, the opportunity to reuse calculation results across users is lessened or eliminated. As a result, the performance benefits associated with reusing the query processor cache are not as high.

Partial expressions (that is, a piece of a calculation that may be used more than once in the expression) and cell properties are not cached. Consider creating a separate calculated member to allow the query processor to cache results when first evaluated and reuse the results in subsequent references. For more information, see Cache Partial Expressions and Cell Properties).

### Query Processor Internals

There are several changes to query processor intervals in SQL Server 2008 Analysis Services. In this section, these changes are discussed before specific optimization techniques are introduced.

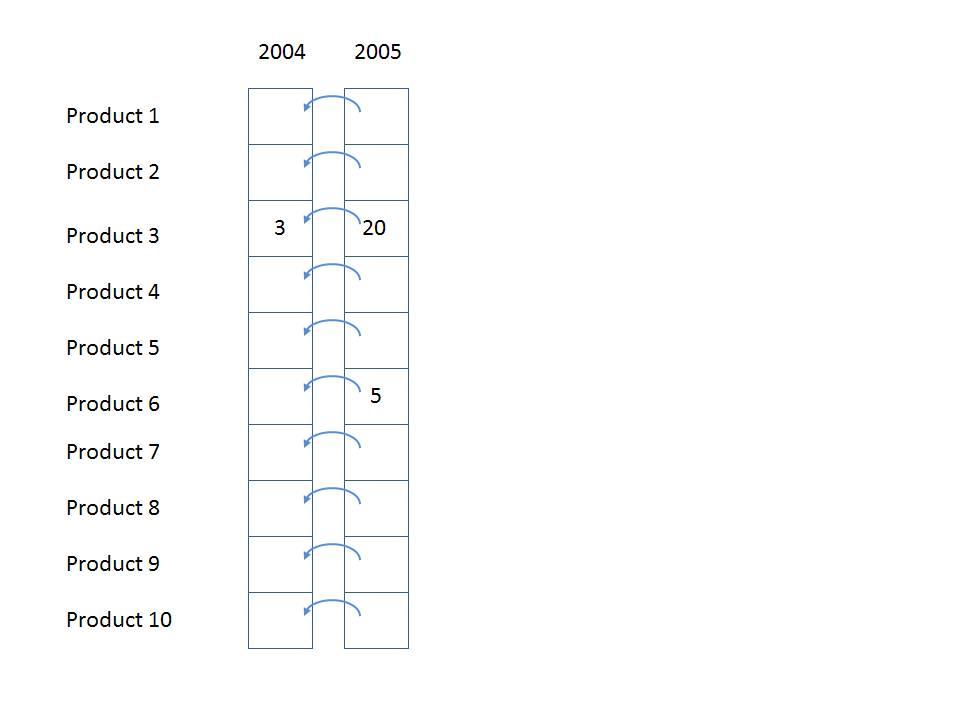
#### Subspace Computation

The key idea behind subspace computation is best introduced by contrasting it with a naïve or cell-by-cell evaluation of a calculation. Consider a trivial calculation RollingSum that sums the sales for the previous year and the current year, and a query that requests the RollingSum for 2005 for all Products.

RollingSum = (Year.PrevMember, Sales) + Sales

SELECT 2005 on columns, Product.Members on rows WHERE RollingSum

A cell-by-cell evaluation of this calculation would then proceed as represented in Figure 4.



**Figure 4 Cell-by -cell evaluation**

The 10 cells for [2005, All Products] would each be evaluated in turn. For each, we would navigate to the previous year, obtain the sales value, and add it to the sales for the current year. There are two significant performance issues with this approach.

Firstly, if the data is *sparse* (that is, thinly populated), then cells are calculated even though they are bound to return a null value. In the example above, calculating the cells for anything but Product 3 and Product 6 is a waste of effort. The impact of this can be extreme – in a sparsely populated cube, the difference can be several orders of magnitude in the numbers of cells evaluated.

Secondly, even if the data is totally *dense*, meaning that every cell has a value and there is no wasted effort visiting empty cells, there is much repeated effort. The same work (for example, getting the previous Year member, setting up the new context for the previous Year cell, checking for recursion) is redone for each Product. It would be much more efficient to move this work out of the inner loop of evaluating each cell.

Now consider the same example performed using subspace computation. Firstly, we can consider that we work our way down an execution tree determining what spaces need to be filled. Given the query, we need to compute the space

[Product.\*, 2005, RollingSum]

(where \* means every member of the attribute hierarchy). Given the calculation, this means we must first compute the space

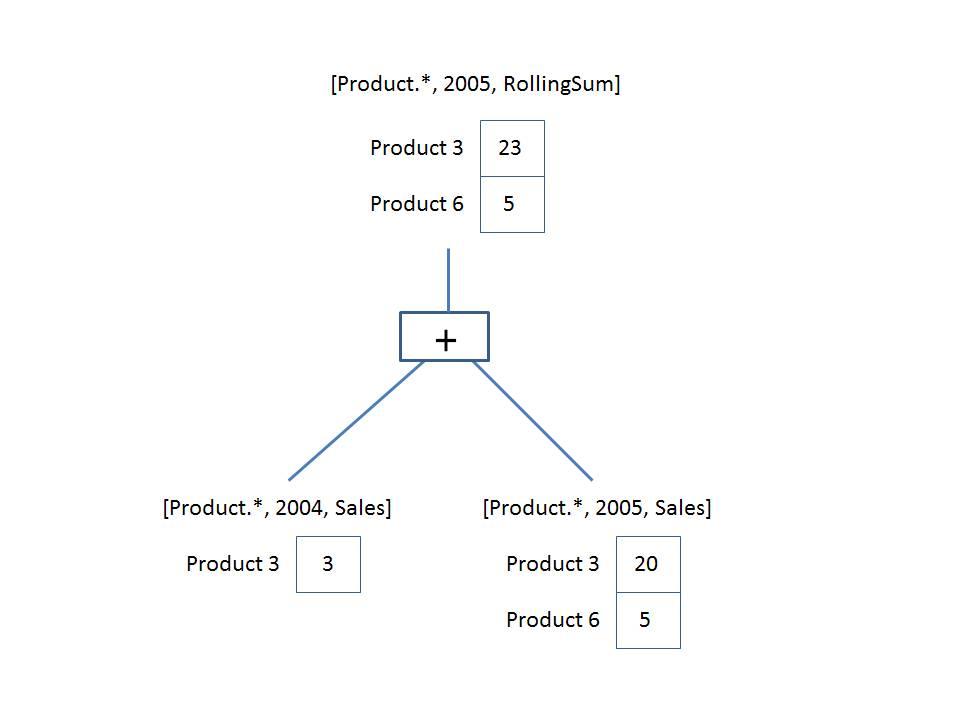
[Product.\*, 2004, Sales]

followed by the space

[Product.\*, 2005, Sales]

and then apply the + operator to those two spaces.

If Sales were itself covered by calculations, then the spaces necessary to calculate Sales would be determined and the tree would be expanded. In this case Sales is a base measure, so we simply obtain the storage engine data to fill the two spaces at the leaves, and then work up the tree, applying the operator to fill the space at the root. Hence the one row (Product3, 2004, 3) and the two rows { (Product3, 2005, 20), (Product6, 2005, 5)} are retrieved, and the + operator applied to them to yields the result in Figure 5.



**Figure 5 Execution plan**

The + operator operates on *spaces*, not simply *scalar* *values.* It is responsible for combining the two given spaces to produce a space that contains each product that appears in either space with the summed value. This is the *query execution plan*. Note that we are only ever operating on data that could contribute to the result. There is no notion of the theoretical space over which we must perform the calculation.

A query execution plan is not one or the other but can contain both subspace and cell-by-cell nodes. Some functions are not supported in subspace mode and the engine falls back to cell-by-cell mode. But even when evaluating an expression in cell-by-cell mode, the engine can return to subspace mode.

#### Expensive vs. Inexpensive Query Plans

It can be costly to build a query plan. In fact, the cost of building an execution plan can exceed the cost of query execution. The Analysis Services engine has a coarse classification scheme – expensive versus inexpensive. A plan is deemed *expensive* if cell-by-cell mode is used or if cube data must be read to build the plan. Otherwise the execution plan is deemed *inexpensive*.

Cube data is used in query plans in several scenarios. Some query plans result in the mapping of one member to another because of MDX functions such as **PrevMember** and **Parent**. The mappings are built from cube data and materialized during the construction of the query plans. The **IIf**, CASE, and IF functions can generate expensive query plans as well should it be necessary to read cube data in order to partition cube space for evaluation of one of the branches. For more information, see IIf Function in SQL Server 2008 Analysis Services.

#### Expression Sparsity

An expression’s *sparsity* refers to the number of cells with nonnull values compared to the total number of cells. If there are relatively few nonnull values, the expression is termed sparse. If there are many, the expression is dense. As we shall see later, whether an expression is sparse or dense can influence the query plan.

But how can you tell if an expression is dense or sparse? Consider a simple noncalculated measure – is it dense or sparse? In OLAP, base fact measures are sparse. This means that typical measure does not have values for every attribute member. For example, a customer does not purchase most products on most days from most stores. In fact it’s the quite the opposite. A typical customer purchases a small percentage of all products from a small number of stores on a few days. There are some other simple rules for popular expressions below.

| Expression | Sparse/dense |
| --- | --- |
| Regular measure | Sparse |
| Constant Value | Dense (excluding constant null values, true/false values) |
| Scalar expression; e.g., count, .properties | Dense |
| <exp1>+<exp2>  <exp1>-<exp2> | Sparse if both exp1 and exp1 are sparse; otherwise dense |
| <exp1>\*<exp2> | Sparse if either exp1 or exp1 is sparse; otherwise dense |
| <exp1> / <exp2> | Sparse if <exp1> is sparse; otherwise dense |
| Sum(<set>, <exp>)  Aggregate(<set>, <exp>) | Inherited from <exp> |
| IIf(<cond>, <exp1>, <exp2>) | Determined by sparsity of default branch (refer to **IIf**) |

#### Default Values

Every expression has a default value – the value the expression assumes most of the time. The query processor calculates an expression’s default value and reuses across most of its space. Most of the time this is null (blank or empty in the Microsoft Excel® spreadsheet software) because oftentimes (but not always) the result of an expression with null input values is null. The engine can then compute the null result once and need only compute values for the much reduced nonnull space.

Another important use of the default values is in the condition in the **IIf** function. Knowing which branch is evaluated more often drives the execution plan. The default values of some popular expressions are listed in the following table.

| Expression | Default value | Comment |
| --- | --- | --- |
| Regular measure | Null | None. |
| IsEmpty(<regular measure>) | True | The majority of theoretical space is occupied by null values. Therefore, **IsEmpty** will return True most often. |
| <regular measure A> = <regular measure B> | True | Values for both measures are principally null, so this will evaluate to True most of the time. |
| <member A> IS <member B> | False | This is different than comparing values – the engine assumes that different members are compared most of the time. |

#### Varying Attributes

Cell values mostly depend on attribute coordinates. But some calculations do not depend on every attribute. For example, the expression

[Customer].[Customer Geography].properties("Postal Code")

depends only on the Customer attribute in the customer dimension. When this expression is evaluated over a subspace involving other attributes, any attributes the expression doesn’t depend on can be eliminated, the expression resolved and projected back over the original subspace. The attributes an expression depends on are termed its varying attributes. For example, consider the following query:

with member measures.Zip as

[Customer].[Customer Geography].currentmember.properties("Postal Code")

select measures.zip on 0,

[Product].[Category].members on 1

from [Adventure Works]

where [Customer].[Customer Geography].[Customer].&[25818]

The expression depends on the customer attribute and not the category attribute; therefore, customer is a varying attribute and category is not. In this case the expression is evaluated only once for the customer and not as many times as there are product categories.

#### Query Processor Internals Wrap-up

Query plans, expression sparsity, default values and varying attributes are core internal concepts behind the query processor behavior – we’ll be returning to these concepts as we discuss optimizing query performance.

## Data Retrieval

When you query a cube, the query processor decomposes the query into subcube requests for the storage engine. For each subcube request, the storage engine first attempts to retrieve data from the storage engine cache. If no data is available in the cache, it attempts to retrieve data from an aggregation. If no aggregation is present, it must retrieve the data from the fact data from a measure group’s partitions.

Each partition is divided in groups of 64K records called a segment.

A coordinator job is created for each subcube request. It creates as many jobs as there are partitions. (This is true where the query requests data within the partition slice. For more information, see Partition Slicing.). Each of these jobs:

* Queues up another job for the next segment (if the current segment is not the last).
* Uses the bitmap indexes to determine if there is data in the segment corresponding to the subcube request.
* Scans the segment, if there is data.

For a single partition, the job structure looks like this after each segment job is queued up.

Immediately after a segment job is queued, it kicks off other segment jobs, and there are as many jobs as there are segments. Should the indexes reveal that no data corresponding to the subcube is contained in the segment, the job ends.

# Enhancing Query Performance

To improve query performance, one must first understand the current situation, diagnose the bottleneck, and then apply one of several techniques including optimizing dimension design, designing and building aggregations, partitioning, and applying best practices.

Much time can be expended pursuing dead ends – it is important to first understand the nature of the problem before applying specific techniques.

## Baselining Query Speeds

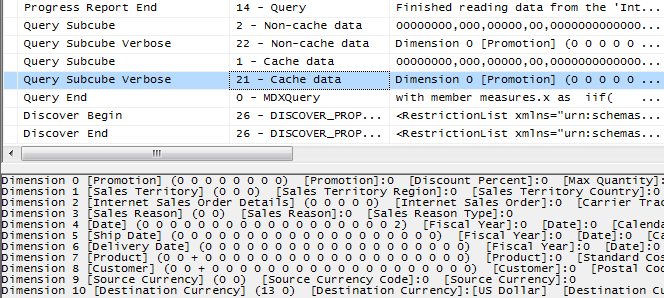
Before beginning optimization, you need a reproducible baseline. Take a measurement on a *cold* (that is, unpopulated) storage engine and query processor caches and a *warm* operating system cache. To do this, execute the query, empty the formula and storage engine caches, and then initialize the calc script by executing a query that returns and caches nothing, as follows.

select {} on 0 from [Adventure Works]

Execute the query a second time. When the query is executed the second time, use [SQL Server Profiler](http://msdn.microsoft.com/en-us/library/ms174779.aspx) to take a trace with the additional events enabled:

* Query Processing\Query Subcube Verbose
* Query Processing\Get Data From Aggregation

The trace contains important information.



**Figure 7 Sample trace**

The text for the query subcube verbose event deserves some explanation. It contains information for each attribute in every dimension:

* 0: Indicates that the attribute is not included in query (the All member is hit).
* \* : Indicates that every member of the attribute was requested.
* + : Indicates that two or more members of the attribute were requested.
* <integer value> : Indicates that a single member of the attribute was hit. The integer represents the member’s data ID (an internal identifier generated by the engine).

Save the trace – it contains important timing information, and it indicates events described later.

To empty the storage and query processor caches, use the ClearCache command.

<ClearCache xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">

<Object>

<DatabaseID>Adventure Works DW</DatabaseID>

</Object>

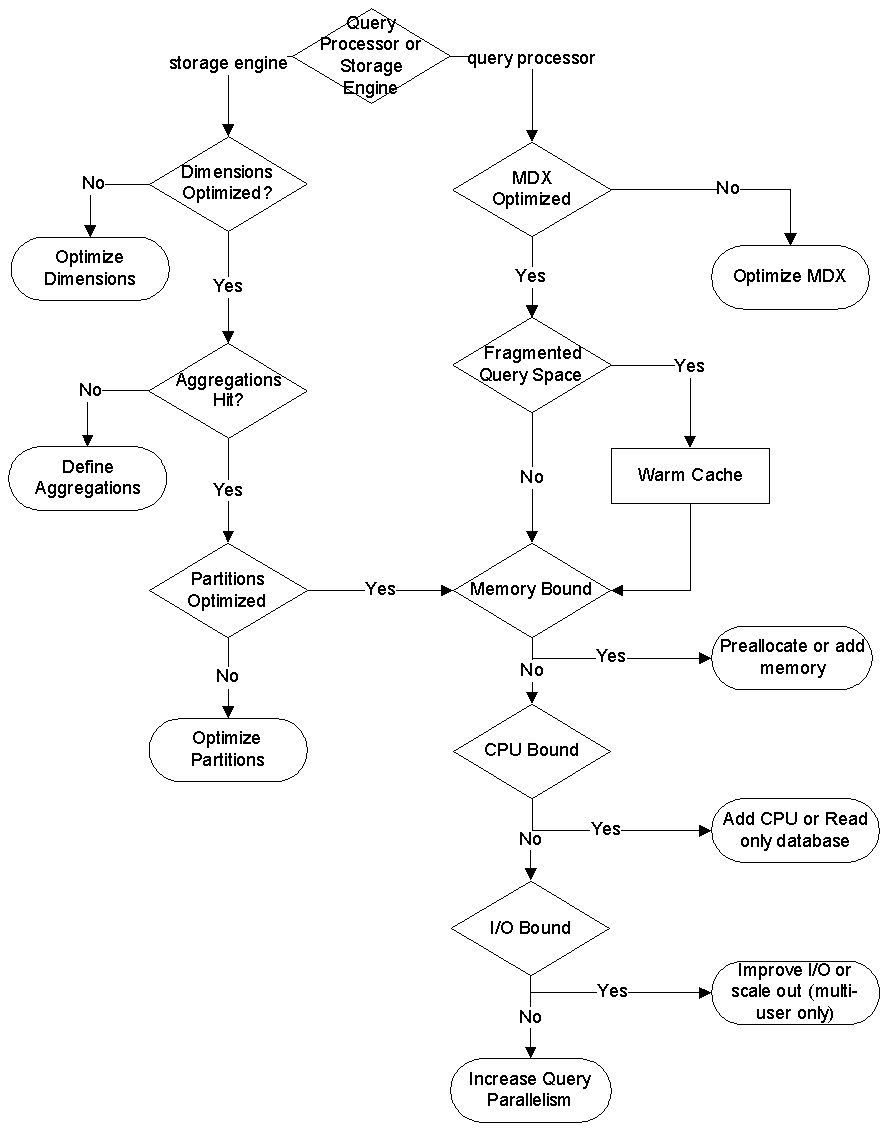
</ClearCache>

The operating system file cache is affected by everything else on the hardware – try to reduce or eliminate other activity. This can be particularly difficult if the cube is stored on a storage area network (SAN) used by other applications.

SQL Server Management Studio reveals query times, but be careful. This time is the amount of time taken to retrieve and display the cellset. For large results, the time to render the cellset can rival the time it took the server to generate it. A SQL Server Profiler trace provides not only insight into where the time is being spent but also the precise engine duration.

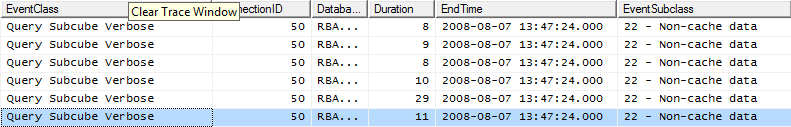
## Diagnosing Query Performance Issues

When performance is not what you expect, the source can be in a number of areas. Figure 8 illustrates how the source of the problem can be diagnosed.



**Figure 8 Query performance tuning flow chart**

The first step is to determine whether the problem lies in the query processor or storage engine. To determine the amount of time the engine is scanning data, use SQL Server Profiler to create a trace. Limit the events to noncached storage engine retrievals by selecting only the query subcube verbose event and filtering on event subclass=22. The result will be similar to Figure 9.

If the majority of time is spent in the storage engine with long running query subcube events, the problem is likely with the storage engine. Consider optimizing dimension design, designing aggregations, or using partitions to improve query performance. If the majority of time is not spent in the storage engine but in the query processor, focus on optimizing MDX. 

The problem can involve both the formula and storage engines. Fragmented query space can be diagnosed with profiler where many query subcube events are generated. Each request may not take long, but the sum of them may. If this is the case, consider warming the cache to reduce the I/O thrashing that this may engender.

Some multiple-user performance issues can be resolved by addressing single-user queries, but certainly not all. Some configuration settings custom to multiple-user environments are described in the section Improving Multiple-User Performance.

If the cube is optimized, CPU and memory resource utilization can be optimized. How to increase the number of threads for single and multiple-user scenarios is described in the section Increasing Query Parallelism. The same technique can be used for reserving memory for improving query and processing performance and is included in the processing section entitled Using PreAllocate.

Performance can generally improved by scaling up with CPU, memory, or I/O. Such recommendations are out of the scope of this document. There are other techniques available to scale out with clusters or read-only databases. These are only described briefly in later sections to determine whether such a path might be the right direction to take.

Monitoring memory usage is discussed in a separate section, Monitoring and Adjusting Server Memory.

## Optimizing Dimensions

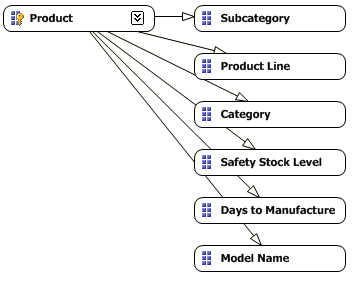
A well-tuned dimension design is one of the most critical success factors of a high-performing Analysis Services solution. One of the first steps to improve cube performance is to step through the dimensions and study attribute relationships. The two most important techniques that you can use to optimize your dimension design for query performance are:

* Identifying attribute relationships.
* Using user hierarchies effectively.

### Identifying Attribute Relationships

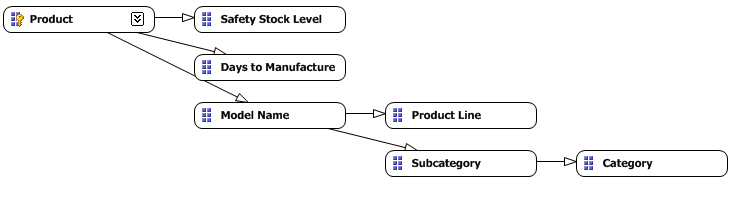
[Attribute relationships](http://msdn2.microsoft.com/en-us/library/ms176124.aspx) define functional dependencies between attributes. In other words, if A has a related attribute B, written A ➔ B, there is one member in B for every member in A, and many members in A for a given member in B. More specifically, given an attribute relationship City ➔ State, if the current city is Seattle, then we know the State must be Washington.

Oftentimes there are relationships between attributes that might or might not be manifested in the original dimension table that can be used by the Analysis Services engine to optimize performance. By default, all attributes are related to the key, and the attribute relationship diagram represents a “bush” where relationships all stem from the key attribute and end at each other’s attribute.



**Figure 10 Bushy attribute relationships**

You can optimize performance by defining relationships supported by the data. In this case, a model name identifies the product line and subcategory, and the subcategory identifies a category (in other words, a single subcategory is not found in more than one category). After redefining the relationships in the attribute relationship editor, we have the following.



**Figure 11 Redefined attribute relationships**

Attribute relationships help performance in two significant ways:

* Indexes are built and cross products need not go through the key attribute.
* Aggregations built on attributes can be reused for queries on related attributes.

Consider the cross-product between Subcategory and Category in the two figures above. In the first - where no attribute relationships have been explicitly defined - the engine must first find which products are in each subcategory and then determine which categories each of these products belongs to. For nontrivially sized dimensions, this can take time. If the attribute relationship is defined, then the Analysis Services engine knows beforehand which category each subcategory belongs to via indexes built at process time.

When defining the attribute relationship, consider the relationship type as flexible or rigid. A flexible attribute relationship is one where members can move around during dimension updates, and a rigid attribute relationship is one where the member relationships are guaranteed to be fixed. For example, the relationship between month and year is fixed because a particular month isn’t going to change its year when the dimension is reprocessed. However, the relationship between customer and city may be flexible as customers move. (As a side note, defining an aggregation to be flexible or rigid has no impact on query performance.)

### Using Hierarchies Effectively

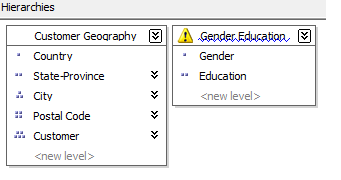
Attributes only exposed in attribute hierarchies are not automatically considered for aggregation by the Aggregation Design Wizard. Queries involving these attributes are satisfied by summarizing data from the primary key. Without the benefit of aggregations, query performance against these attributes hierarchies can be slow.

To enhance performance, it is possible to flag an attribute as an aggregation candidate by using the **Aggregation Usage** property. For more detailed information on this technique, see Suggesting Aggregation Candidates. However, before you modify the **Aggregation Usage** property, you should consider whether you can take advantage of user hierarchies.

Analysis Services enables you to build two types of user hierarchies: natural and unnatural hierarchies, each with different design and performance characteristics.

In a *natural hierarchy*, all attributes participating as levels in the hierarchy have direct or indirect attribute relationships from the bottom of the hierarchy to the top of the hierarchy.

In an *unnatural hierarchy,* the hierarchy consists of at least two consecutive levels that have no attribute relationships. Typically these hierarchies are used to create drill-down paths of commonly viewed attributes that do not follow any natural hierarchy. For example, users may want to view a hierarchy of Gender and Education.



**Figure 12 Natural and unnatural hierarchies**

From a performance perspective, natural hierarchies behave very differently than unnatural hierarchies. In natural hierarchies, the hierarchy tree is materialized on disk in hierarchy stores. In addition, all attributes participating in natural hierarchies are automatically considered to be aggregation candidates.

Unnatural hierarchies are not materialized on disk, and the attributes participating in unnatural hierarchies are not automatically considered as aggregation candidates. Rather, they simply provide users with easy-to-use drill-down paths for commonly viewed attributes that do not have natural relationships. By assembling these attributes into hierarchies, you can also use a variety of MDX navigation functions to easily perform calculations like percent of parent.

To take advantage of natural hierarchies, define cascading attribute relationships for all attributes participating in the hierarchy.

## Maximizing the Value of Aggregations

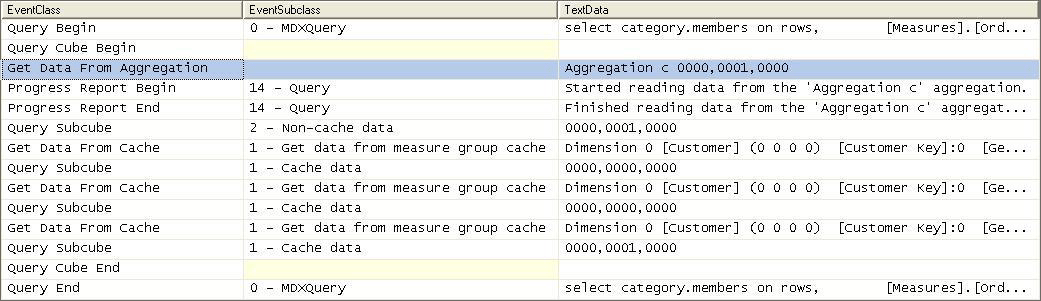
An *aggregation* is a precalculated summary of data that Analysis Services uses to enhance query performance.

Designing aggregations is the process of selecting the most effective aggregations for your querying workload. As you design aggregations, you must consider the querying benefits that aggregations provide compared with the time it takes to create and refresh the aggregations. In fact, adding unnecessary aggregations can worsen query performance because the rare hits move the aggregation into the file cache at the cost of moving something else out.

While aggregations are physically designed per measure group partition, the optimization techniques for maximizing aggregation design apply whether you have one or many partitions. In this section, unless otherwise stated, aggregations are discussed in the fundamental context of a cube with a single measure group and single partition. For more information on how you can improve query performance using multiple partitions, see Using Partitions to Enhance Query Performance.

### Detecting Aggregation Hits

Use SQL Server Profiler to view how and when aggregations are used to satisfy queries. Within SQL Server Profiler, there are several events that describe how a query is fulfilled. The event that specifically pertains to aggregation hits is the **Get Data From Aggregation** event.



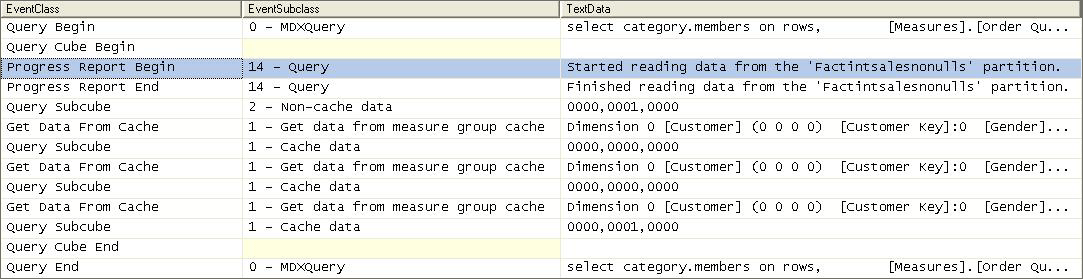
**Figure 13 Scenario 1: SQL Server Profiler trace for cube with an aggregation hit**

Figure 13 displays a SQL Server Profiler trace of the query’s resolution against a cube with aggregations. In the SQL Server Profiler trace, the operations that the storage engine performs to produce the result set are revealed.

The storage engine gets data from Aggregation C 0000, 0001, 0000 as indicated by the **Get Data From Aggregation** event. In addition to the aggregation name, Aggregation C, Figure 13 displays a vector, **000, 0001, 0000**, that describes the content of the aggregation. More information on what this vector actually means is described in the next section, How to Interpret Aggregations.

The aggregation data is loaded into the storage engine measure group cache from where the query processor retrieves it and returns the result set to the client.

Figure 14 displays a SQL Server Profiler trace for the same query against the same cube, but this time, the cube has no aggregations that can satisfy the query request.



**Figure 14 Scenario 2: SQL Server Profiler trace for cube with no aggregation hit**

After the query is submitted, rather than retrieving data from an aggregation, the storage engine goes to the detail data in the partition. From this point, the process is the same. The data is loaded into the storage engine measure group cache.

### How to Interpret Aggregations

When Analysis Services creates an aggregation, each dimension is named by a vector, indicating whether the attribute points to the attribute or to the All level. The Attribute level is represented by 1 and the All level is represented by 0. For example, consider the following examples of aggregation vectors for the product dimension:

* **Aggregation By ProductKey Attribute** = [Product Key]:1 [Color]:0 [Subcategory]:0 [Category]:0 or **1000**
* **Aggregation By Category Attribute** = [Product Key]:0 [Color]:0 [Subcategory]:0 [Category]:1 or **0001**
* **Aggregation By ProductKey.All** and **Color.All** and **Subcategory.All** and **Category.All** = [Product Key]:0 [Color]:0 [Subcategory]:0 [Category]:0 or **0000**

To identify each aggregation, Analysis Services combines the dimension vectors into one long vector path, also called a *subcube*, with each dimension vector separated by commas.

The order of the dimensions in the vector is determined by the order of the dimensions in the cube. To find the order of dimensions in the cube, use one of the following two techniques. With the cube opened in SQL Server Business Intelligence Development Studio, you can review the order of dimensions in a cube on the **Cube Structure** tab. The order of dimensions in the cube is displayed in the Dimensions pane. As an alternative, you can review the order of dimensions listed in the cube’s XMLA definition.

The order of attributes in the vector for each dimension is determined by the order of attributes in the dimension. You can identify the order of attributes in each dimension by reviewing the dimension XML file.

For example, the following subcube definition (0000, 0001, 0001) describes an aggregation for the following:

* Product – All, All, All, All
* Customer – All, All, All, State/Province
* Order Date – All, All, All, Year

Understanding how to read these vectors is helpful when you review aggregation hits in SQL Server Profiler. In SQL Server Profiler, you can view how the vector maps to specific dimension attributes by enabling the **Query Subcube Verbose** event.

### Building Aggregations

To help Analysis Services successfully apply the aggregation design algorithm, you can perform the following optimization techniques to influence and enhance the aggregation design. (The sections that follow describe each of these techniques in more detail).

**Suggesting aggregation candidates** – When Analysis Services designs aggregations, the aggregation design algorithm does not automatically consider every attribute for aggregation. Consequently, in your cube design, verify the attributes that are considered for aggregation and determine whether you need to suggest additional aggregation candidates.

**Specifying statistics about cube data** –To make intelligent assessments of aggregation costs, the design algorithm analyzes statistics about the cube for each aggregation candidate. Examples of this metadata include member counts and fact table counts. Ensuring that your metadata is up-to-date can improve the effectiveness of your aggregation design.

**Usage-based optimization** – To focus aggregations on particular usage pattern, execute the queries and launch the Usage-Based Optimization Wizard.

#### Suggesting Aggregation Candidates

When Analysis Services designs aggregations, the aggregation design algorithm does not automatically consider every attribute for aggregation. To streamline this process, Analysis Services uses the **Aggregation Usage** property to determine which attributes it should consider. For every measure group, verify the attributes that are automatically considered for aggregation and then determine whether you need to suggest additional aggregation candidates.

**Aggregation Usage Rules**

An *aggregation candidate* is an attribute that Analysis Services considers for potential aggregation. To determine whether or not a specific attribute is an aggregation candidate, the storage engine relies on the value of the **Aggregation Usage** property. The **Aggregation Usage** property is assigned a per-cube attribute, so it globally applies across all measure groups and partitions in the cube. For each attribute in a cube, the **Aggregation Usage** property can have one of four potential values: **Full**, **None**, **Unrestricted**, and **Default**.

**Full**— Every aggregation for the cube must include this attribute or a related attribute that is lower in the attribute chain. For example, you have a product dimension with the following chain of related attributes: Product, Product Subcategory, and Product Category. If you specify the **Aggregation Usage** for Product Category to be **Full**, Analysis Services may create an aggregation that includes Product Subcategory as opposed to Product Category, given that Product Subcategory is related to Category and can be used to derive Category totals.

**None**—No aggregation for the cube can include this attribute.

**Unrestricted**—No restrictions are placed on the aggregation designer; however, the attribute must still be evaluated to determine whether it is a valuable aggregation candidate.

**Default**—The designer applies a *default rule* based on the type of attribute and dimension. This is the default value of the **Aggregation Usage** property.

The default rule is highly conservative about which attributes are considered for aggregation. The default rule is broken down into four constraints.

**Default Constraint 1—Unrestricted** - For a dimension’s measure group granularity attribute, default means **Unrestricted**. The granularity attribute is the same as the dimension’s key attribute as long as the measure group joins to a dimension using the primary key attribute.

**Default Constraint 2**—**None for Special Dimension Types -** For all attributes (except All) in many-to-many, nonmaterialized reference dimensions, and data mining dimensions, default means **None**.

**Default Constraint 3**—**Unrestricted for Natural Hierarchies -** A naturalhierarchy is a user hierarchy where all attributes participating in the hierarchy contain attribute relationships to the attribute sourcing the next level. For such attributes, default means **Unrestricted,** except for nonaggregatable attributes, which are set to **Full** (even if they are not in a user hierarchy).

**Default Constraint 4**—**None For Everything Else**. For all other dimension attributes, default means **None**.

#### Influencing Aggregation Candidates

In light of the behavior of the **Aggregation Usage** property, use the following guidelines:

**Attributes exposed solely as attribute hierarchies**- If a given attribute is only exposed as an attribute hierarchy such as Color, you may want to change its **Aggregation Usage** property as follows.

First, change the value of the **Aggregation Usage** property from **Default** to **Unrestricted** if the attribute is a commonly used attribute or if there are special considerations for improving the performance in a particular pivot or drilldown. For example, if you have highly summarized scorecard style reports, you want to ensure that the users experience good initial query response time before drilling around into more detail.

While setting the **Aggregation Usage** property of a particular attribute hierarchy to **Unrestricted** is appropriate is some scenarios, do not set all attribute hierarchies to **Unrestricted**. Increasing the number of attributes to be considered increases the problem space the aggregation algorithm must consider. The wizard can take at least an hour to complete the design and considerably much more time to process. Set the property to **Unrestricted** only for the commonly queried attribute hierarchies. The general rule is five to ten **Unrestricted** attributes per dimension.

Next, change the value of the **Aggregation Usage** property from **Default** to **Full** in the unusual case that it is used in virtually every query you want to optimize. This is a rare case, and this change should be made only for attributes that have a relatively small number of members.

**Infrequently used attributes**—For attributes participating in natural hierarchies, you may want to change the **Aggregation Usage** property from **Default** to **None** if users would only infrequently use it. Using this approach can help you reduce the aggregation space and get to the five to ten **Unrestricted** attributes per dimension. For example, you may have certain attributes that are only used by a few advanced users who are willing to accept slightly slower performance. In this scenario, you are essentially forcing the aggregation design algorithm to spend time building only the aggregations that provide the most benefit to the majority of users.

The aggregation design algorithm evaluates the cost/benefit of each aggregation based member counts and fact table record counts. Ensuring that your metadata is up-to-date can improve the effectiveness of your aggregation design. You can define the fact table source record count in the **EstimatedRows** property of each measure group, and you can define attribute member count in the **EstimatedCount** property of each attribute.

#### Usage-Based Optimization

The Usage-Based Optimization Wizard reviews the queries in the query log (something you must set up beforehand) and designs aggregations that cover the top 100 slowest queries. Use the Usage-Based Optimization Wizard with a 100% performance gain - this will design aggregations to avoid hitting the partition directly.

After the aggregations are designed, you can add them to the existing design or completely replace the design. Be careful adding them to the existing design – the two designs may contain aggregations that serve almost identical purposes that when combined are redundant with one another. Inspect the new aggregations compared to the old and ensure there are no near-duplicates. The aggregation design can be copied to other partitions in SQL Server Management Studio or Business Intelligence Design Studio.

Aggregation designs have a costly metadata impact – don’t overdesign but try to keep the number of aggregation designs per measure group to a minimum.

#### Aggregations and Parent-Child Hierarchies

In parent-child hierarchies, aggregations are created only for the key attribute and the top attribute, i.e., the All attribute unless it is disabled. Refrain from using parent-child hierarchies that contain a large number of members. (How big is large? There isn’t a specific number because query performance at intermediate levels of the parent-child hierarchy will degrade linearly with the number of members.) Additionally, limit the number of parent-child hierarchies in your cube.

If you are in a design scenario with a large parent-child hierarchy, consider altering the source schema to reorganize part or all of the hierarchy into a regular hierarchy with a fixed number of levels. After the data has been reorganized into the user hierarchy, you can use the **Hide Member If** property of each level to hide the redundant or missing members.

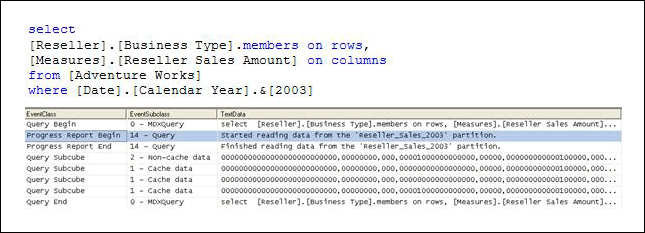
## Using Partitions to Enhance Query Performance

Partitions separate measure group data into physical units. Effective use of partitions can enhance query performance, improve processing performance, and facilitate data management. This section specifically addresses how you can use partitions to improve query performance. You must balance the benefits and costs between query and processing performance before you finalize your partitioning strategy.

### Introduction

You can use multiple partitions to break up your measure group into separate physical components. The advantages of partitioning for improving query performance are:

* Partition slicing: Partitions not containing data in the subcube are not queried at all, thus avoiding the cost of reading the index (or scanning the table in ROLAP mode, where there are no MOLAP indexes).
* Aggregation design: Each partition can have its own or shared aggregation design. Therefore, partitions queried more often or differently can have their own designs.



**Figure 15 Intelligent querying by partitions**

Figure 15 displays the profiler trace of query requesting Reseller Sales Amount by Business Type from Adventure Works. The Reseller Sales measure group of the Adventure Works cube contains four partitions: one for each year. Because the query slices on 2003, the storage engine can go directly to the 2003 Reseller Sales partition and ignore other partitions.

### Partition Slicing

Partitions are bound to a source table, view, or source query. For MOLAP partitions, during processing Analysis Services internally identifies the range of data that is contained in each partition by using the Min and Max DataIDs of each attribute to calculate the range of data that is contained in the partition. The data range for each attribute is then combined to create the slice definition for the partition. Knowing this information, the storage engine can optimize which partitions it scans during querying by only choosing those partitions that are relevant to the query. For ROLAP and proactive caching partitions, you must manually identify the slice in the properties of the partition.

The Min and Max DataIDs can specify a single member or a range. For example, partitioning by year results in the same Min and Max DataID slice for the year attribute, and queries to a specific moment in time only result in partition queries to that year’s partition.

It is important to remember that the partition slice is maintained as a range of DataIDs that you have no explicit control over. DataIDs are assigned during dimension processing as new members are encountered. If they are out of order in the dimension table, then the internal sequence of DataIDs can differ from attribute keys. This can cause unnecessary partition reads. For this reason, there may be a benefit to define the slice yourself for MOLAP partitions. For example, if you partition by year with some partitions containing a range of years, defining the slice explicitly avoids the problem of overlapping DataIDs.

Whenever you use multiple partitions for a given measure group, ensure that you update the data statistics for each partition. More specifically, it is important to ensure that the partition data and member counts accurately reflect the specific data in the partition and not the data across the entire measure group.

Note that the slice is not defined and indexes are not built for partitions with fewer rows than **IndexBuildThreshold** (which has a default value of 4096).

### Aggregation Considerations for Multiple Partitions

When you define your partitions, remember that they do not have to contain uniform datasets nor aggregation designs. For example, for a given measure group, you may have 3 yearly partitions, 11 monthly partitions, 3 weekly partitions, and 1–7 daily partitions. The value of using heterogeneous partitions with different levels of detail is that you can more easily manage the loading of new data without disturbing existing partitions (more on this in the processing section) and you can design aggregations for groups of partitions that share the same level of detail.

For each partition, you can use a different aggregation design. By taking advantage of this flexibility, you can identify those data sets that require higher aggregation design.

Consider the following example. In a cube with multiple monthly partitions, new data may flow into the single partition corresponding to the latest month. Generally that is also the partition most frequently queried. A common aggregation strategy in this case is to perform usage-based optimization to the most recent partition, leaving older, less frequently queried partitions as they are.

The newest aggregation design can also be copied to a *base partition*. This base partition holds no data—it serves only to hold the current aggregation design. When it is time to add a new partition (for example, at the start of a new month), the base partition can be cloned to a new partition. When the slice is set on the new partition, it is ready to take data as the current partition. Following an initial full process, the current partition can be incrementally updated for the remainder of the period.

### Distinct Count Partition Design

Distinct count partitions are special. When distinct count partitions are queried, each partition’s segment jobs must coordinate with one another to avoid counting duplicates. For example, if counting distinct customers with customer ID and the same customer ID is in multiple partitions, the partitions’ jobs must recognize the match to not count the same customer more than once.

If each partition contains nonoverlapping range of values, this coordination between jobs is avoided and query performance can improve by between 20 to 300 percent!

For more information about optimizations for distinct count, see the white paper “Analysis Services Distinct Count”, which is available from the following link: <http://www.microsoft.com/downloads/details.aspx?FamilyID=65df6ebf-9d1c-405f-84b1-08f492af52dd&displaylang=en>

### Partition Sizing

For nondistinct count measure groups, tests with partition sizes in the range of 200 megabytes (MB) to up to 3 gigabytes (GB) indicate that partition size alone does not have a substantial impact on query speeds. The partitioning strategy should be based on these factors:

* Increasing processing speed and flexibility
* Increasing manageability of bringing in new data
* Increasing query performance from partition elimination
* Support for different aggregation designs

## Optimizing MDX

Debugging calculation performance issues across a cube can be difficult if there are many calculations. The first step is to try to narrow down where the problem expression is and then apply best practices to the MDX.

### Diagnosing the Problem

Diagnosing the problem may be straightforward if a simple query calls out a specific calculation (in which case continue to the next section,) but if there are chains of expressions or a complex query, it can be time-consuming to locate the problem.

Try to reduce the query to simplest expression possible that continues to reproduce the performance issue. With some client applications, the query itself can be problem, should it demand large data volumes, push down to unnecessarily low granularities (bypassing aggregations), or contain query calculations that bypass the global and session query processor caches.

If the issue is confirmed to be in the cube itself, remove or comment out all calculations from the cube. This includes the following:

* Custom member formulas
* Unary operators
* MDX scripts (except the calculate statement, which should be left intact)

Rerun the query. It might have to be altered to account for missing members. Bring back the calculations until the problem is reproduced.

### Calculation Best Practices

This section contains a series of best practices to apply to get the best query performance from your cube.

#### Cell-by-Cell Mode vs. Subspace Mode

Almost always, performance obtained by using subspace mode is superior to that obtained by using cell-by-cell mode. For more information, including the list of functions supported in subspace mode, see “Performance Improvements for MDX in SQL Server 2008 Analysis Services” in SQL Server Books Online, available at the following link:

<http://msdn.microsoft.com/en-us/library/bb934106(SQL.100).aspx>

The following table lists the most common reasons for leaving subspace mode.

| **Feature or function** | **Comment** |
| --- | --- |
| Set aliases | Replace with a set expression rather than an alias. For example, this query operates in subspace mode:  with  member measures.SubspaceMode as  sum(  [Product].[Category].[Category].members,  [Measures].[Internet Sales Amount]  )  select  {measures.SubspaceMode,[Measures].[Internet Sales Amount]} on 0 ,  [Customer].[Customer Geography].[Country].members on 1  from [Adventure Works]  cell properties value  but almost the same query ,where we replace the set with an alias, operates in cell-by-cell mode:  with  set y as [Product].[Category].[Category].members  member measures.Naive as  sum(  y,  [Measures].[Internet Sales Amount]  )  select  {measures.Naive,[Measures].[Internet Sales Amount]} on 0 ,  [Customer].[Customer Geography].[Country].members on 1  from [Adventure Works]  cell properties value |
| Late binding in functions:  **LinkMember**, **StrToSet**, **StrToMember**, **StrToValue** | Late-binding functions are functions that depend on query context and cannot be statically evaluated. For example, the following code is statically bound:  with member measures.x as (strtomember("[Customer].[Customer Geography].[Country].&[Australia]"),[Measures].[Internet Sales Amount])  select measures.x on 0,  [Customer].[Customer Geography].[Country].members on 1 from [Adventure Works]  cell properties value  A query is late-bound if an argument can be evaluated only in context:  with member measures.x as (strtomember([Customer].[Customer Geography].currentmember.uniquename),[Measures].[Internet Sales Amount])  select measures.x on 0,  [Customer].[Customer Geography].[Country].members on 1 from [Adventure Works]  cell properties value |
| User-defined stored procedures | Popular Microsoft Visual Basic® for Applications (VBA) and Excel functions are natively supported in MDX. User-defined stored procedures are evaluated in cell-by-cell mode. |
| **LookupCube** | Linked measure groups are often a viable alternative. |

#### IIf Function in SQL Server 2008 Analysis Services

The **IIf** MDX function is a commonly used expression that can be costly to evaluate. The engine optimizes performance based on a few simple criteria. The **IIf** function takes three arguments:

iif(<condition>, <then branch>, <else branch>)

Where the condition evaluates to true, the value from the then branch is used; otherwise the else branch expression is used.

Note the term “used” – one or both branches may be evaluated even if its value is not used. It may be cheaper for the engine to evaluate the expression over the entire space and use it when needed - termed an *eager* plan – that it would be to chop up the space into a potentially enormous number of fragments and evaluate only where needed - a *strict* plan.

**Note:** One of the most common errors in MDX scripting is using **IIf** when the condition depends on cell coordinates instead of values. If the condition depends on cell coordinates, use scopes and assignments. When this is done, the condition is not evaluated over the space and the engine does not evaluate one or both branches over the entire space. Admittedly, in some cases, using assignments forces some unwieldy scoping and repetition of assignments, but it is always worthwhile comparing the two approaches.

The first consideration is whether the query plan is expensive or inexpensive. Most **IIf** condition query plans are inexpensive, but complex nested conditions with more **IIf** functions can go to cell by cell.

The next consideration the engine makes is what value the condition takes most. This is driven by the condition’s default value. If the condition’s default value is true, then the then branch is the default branch – the branch that is evaluated over most of the subspace. Knowing a few simple rules on how the condition is evaluated helps to determine the default branch:

* In sparse expressions, most cells are empty. The default value of the **IsEmpty** function on a sparse expression is true.
* Comparison to zero of a sparse expression is true.
* The default value of the IS operator is false.
* If the condition cannot be evaluated in subspace mode, there is no default branch.

For example, one of the most common uses of the **IIf** function is to check whether the denominator is nonzero:

iif([Measures].[Internet Sales Amount]=0, null, [Measures].[Internet Order Quantity]/[Measures].[Internet Sales Amount])

There is no calculation on Internet Sales Amount, so it is sparse. Therefore the default value of the condition is true and therefore the default branch is the then branch with the null expression.

The following table shows how each branch of an **IIf** function is evaluated.

| **Branch query plan** | **Branch is default branch** | **Branch expression sparsity** | **Evaluation** |
| --- | --- | --- | --- |
| Expensive | Not applicable | Not applicable | Strict |
| Inexpensive | True | Not applicable | Eager |
| Inexpensive | False | Dense | Strict |
| Inexpensive | False | Sparse | Eager |

In SQL Server 2008 Analysis Services, you can overrule the default behavior with query hints.

iif(

[<condition>

, <then branch> [hint [Eager | Strict]]

, <else branch> [hint [Eager | Strict]]

)

When would you want to override the default behavior? Here are the most common scenarios where you might want to change the default behavior:

* The engine determines the query plan for the condition is expensive and evaluates each branch in strict mode.
* The condition is evaluated in cell-by-cell mode, and each branch is evaluated in eager mode.
* The branch expression is dense but easily evaluated.

For example, consider the simple expression below taking the inverse of a measure.

with member

measures.x as

iif(

[Measures].[Internet Sales Amount]=0

, null

, (1/[Measures].[Internet Sales Amount]) )

select {[Measures].x} on 0,

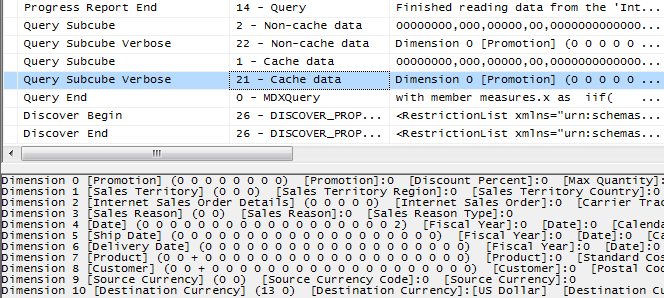
[Customer].[Customer Geography].[Country].members \*

[Product].[Product Categories].[Category].members on 1

from [Adventure Works]

cell properties value

The query plan is not expensive, the else branch is not the default branch, and the expression is dense, so it is evaluated in strict mode. This forces the engine to materialize the space over which it is evaluated. This can be seen in SQL Server Profiler with query subcube verbose events selected as displayed in Figure 16.



**Figure 16 Default IIf query trace**

Note the subcube definition for the Product and Customer dimension (dimensions 7 and 8 respectively) with the ‘+’ indicator on the Country and Category attributes. This means that more than one but not all members are included – the query processor has determined which tuples meet the condition and partitioned the space, and it is evaluating the fraction over that space.

To prevent the query plan from partitioning the space, the query can be modified as follows (in bold).

with member

measures.x as

iif(

[Measures].[Internet Sales Amount]=0

, null

, (1/[Measures].[Internet Sales Amount]) **hint eager**)

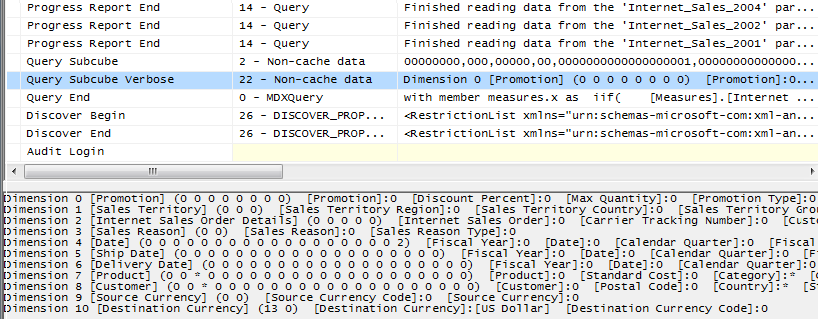
select {[Measures].x} on 0,

[Customer].[Customer Geography].[Country].members \*

[Product].[Product Categories].[Category].members on 1

from [Adventure Works]

cell properties value



**Figure 17 IIf trace with MDX query hints**

Now the same attributes are marked with a ‘\*’ indicator, meaning that the expression is evaluated over the entire space instead of a partitioned space.

#### Cache Partial Expressions and Cell Properties

Partial expressions (those that are part of a calculated member or assignment) are not cached. So if an expensive subexpression is used more than once, consider creating a separate calculated member to allow the query processor to cache and reuse. For example, consider the following.

this = iif(<expensive expression >= 0, 1/<complex expression>, null);

The repeated partial expressions can be extracted and replaced with a hidden calculated member as follows.

create member currentcube.measures.MyPartialExpression as <expensive expression> , visible=0;

this = iif(measures.MyPartialExpression >= 0, 1/ measures.MyPartialExpression, null);

Only the value cell property is cached. If you have complex cell properties to support such things as bubble-up exception coloring, consider creating a separate calculated measure; for example, instead of

create member currentcube.measures.[Value] as <exp> , backgroundColor=<complex expression>;

do this

create member currentcube.measures.MyCellPrope as <complex expression> , visible=0;

create member currentcube.measures.[Value] as <exp> , backgroundColor=<complex expression>;

#### Avoid Mimicking Engine Features with Expressions

Several native features can be mimicked with MDX:

* Unary operators
* Calculated columns in the data source view (DSV)
* Measure expressions
* Semiadditive measures

You can reproduce each these features in MDX script (in fact, sometimes you must, because some are only supported in the Enterprise SKU), but doing so often hurts performance.

For example, distributive unary operators (that is, one whose member order does not matter, such as +, -, and ~) are generally twice as fast as trying to mimic their capabilities with assignments.

There are rare exceptions. For example, one might be able to improve performance of nondistributive unary operators (those involving \*, /, or numeric values) with MDX. Furthermore, you may know some special characteristic of your data that allows you to take a shortcut that improves performance.

#### Eliminate Varying Attributes in Set Expressions

Set expressions do not support varying attributes. This impacts all set functions including **Filter**, **Aggregate**, **Avg**, and others. You can work around this problem by explicitly overwriting invariant attributes to a single member.

For example, in this calculation, the average of sales only including those exceeding $100 is computed.

with member measures.AvgSales as

avg(

filter(

descendants([Customer].[Customer Geography].[All Customers],,leaves)

, [Measures].[Internet Sales Amount]>100

)

,[Measures].[Internet Sales Amount]

)

select measures.AvgSales on 0,

[Customer].[Customer Geography].[City].members on 1

from [Adventure Works]

This takes 2:29 on a laptop – quite a while. However, the average of sales for all customers everywhere does not depend on the current city (this is just another way of saying that city is not a varying attribute). We can explicitly eliminate city as a varying attribute by overwriting it to the all member as follows.

with member measures.AvgSales as

avg(

filter(

descendants([Customer].[Customer Geography].[All Customers],,leaves)

, [Measures].[Internet Sales Amount]>100

)

,[Measures].[Internet Sales Amount]

)

member measures.AvgSalesWithOverWrite as (measures.AvgSales, root([Customer]))

select measures.AvgSalesWithOverWrite on 0,

[Customer].[Customer Geography].[City].members on 1

from [Adventure Works]

This takes less than a second – a substantial change in performance.

#### Avoid Assigning Nonnull Values to Otherwise Nonempty Cells

The Analysis Services engine is very efficient at eliminating empty rows. Adding calculations with nonempty values replacing null values does not allow AS to eliminate these rows. For example, this query replaces null values with the dash, and the **non empty** key word does not eliminate them.

with member measures.x as

iif( not isempty([Measures].[Internet Sales Amount]),[Measures].[Internet Sales Amount],"-")

select descendants([Date].[Calendar].[Calendar Year].&[2004] ) on 0,

non empty [Customer].[Customer Geography].[Customer].members on 1

from [Adventure Works]

where measures.x

**non empty** operates on cell values and not on formatted values. In rare cases we can instead use the format string to replace null values with the same character while still eliminating empty rows and columns in roughly half the time.

with member measures.x as

[Measures].[Internet Sales Amount], FORMAT\_STRING = "#.00;(#.00);#.00;-"

select descendants([Date].[Calendar].[Calendar Year].&[2004] ) on 0,

non empty [Customer].[Customer Geography].[Customer].members on 1

from [Adventure Works]

where measures.x

The reason this can only be used in rare cases is that the query is not equivalent – the second query eliminates completely empty rows. More importantly, neither Excel nor Reporting Services supports the fourth argument in the format\_string. For more information about using the format\_string calculation property, see “FORMAT\_STRING Contents (MDX)” in SQL Server Books Online, which is available at the following link:

<http://msdn.microsoft.com/en-us/library/ms146084.aspx>

#### Eliminate Cost of Computing Formatted Values

In some circumstances, the cost of determining the format string for an expression outweighs the cost of the value itself. To determine if this applies to a slow-running query, compare execution times with and without the formatted value cell property, as in the following query.

select [Measures].[Internet Average Sales Amount] on 0 from [Adventure Works] cell properties value

If the result is noticeable faster without the formatting, apply the formatting directly in the script as follows.

scope([Measures].[Internet Average Sales Amount]);

FORMAT\_STRING(this) = "currency";

end scope;

And execute the query (with formatting applied) to determine the extent of any performance benefit.

#### Sparse/Dense Considerations with “expr1 \* expr2” Expressions

When writing expressions as products of two other expressions, place the sparser one on the left-hand side.

Consider the following two queries, which have the signature of a currency conversion calculation of applying the exchange rate at leaves of the date dimension in Adventure Works. The only difference is exchanging the order of the expressions in the product of the cell calculation. The results are the same, but using the sparser internet sales amount first results in about a 10% savings. (That’s not much in this case, but it could be substantially more in others. Savings depends on relative sparsity between the two expressions, and performance benefits may vary).

**Sparse First**

with cell CALCULATION x for '({[Measures].[Internet Sales Amount]},leaves([Date]))'

as

[Measures].[Internet Sales Amount] \*

([Measures].[Average Rate],[Destination Currency].[Destination Currency].&[EURO])

select

non empty [Date].[Calendar].members on 0,

non empty [Product].[Product Categories].members on 1

from [Adventure Works]

where ([Measures].[Internet Sales Amount], [Customer].[Customer Geography].[State-Province].&[BC]&[CA])

**Dense First**

with cell CALCULATION x for '({[Measures].[Internet Sales Amount]},leaves([Date]))'

as

([Measures].[Average Rate],[Destination Currency].[Destination Currency].&[EURO])\*

[Measures].[Internet Sales Amount]

select

non empty [Date].[Calendar].members on 0,

non empty [Product].[Product Categories].members on 1

from [Adventure Works]

where ([Measures].[Internet Sales Amount], [Customer].[Customer Geography].[State-Province].&[BC]&[CA])

#### Comparing Objects and Values

When determining whether the current member or tuple is a specific object, use IS. For example, the following query is not only nonperformant but incorrect. It forces unnecessary cell evaluation and compares values instead of members.

[Customer].[Customer Geography].[Country].&[Australia] = [Customer].[Customer Geography].currentmember

Furthermore, don’t perform extra steps when deducing whether **CurrentMember** is a particular member by involving **Intersect** and **Counting**.

intersect({[Customer].[Customer Geography].[Country].&[Australia]}, [Customer].[Customer Geography].currentmember).count > 0

Do this.

[Customer].[Customer Geography].[Country].&[Australia] is [Customer].[Customer Geography].currentmember

#### Evaluating Set Membership

Determining whether a member or tuple is in a set is best accomplished with **Intersect**. The **Rank** function does the additional operation of determining where in the set that object lies. If you don’t need it, don’t use it. For example, instead of this

rank( [Customer].[Customer Geography].[Country].&[Australia],

<set expression> )>0

Do this

intersect({[Customer].[Customer Geography].[Country].&[Australia]}, <set> ).count > 0

#### Consider Moving Calculations to the Relational Engine

Sometimes calculations can be moved to the Relational Engine and be processed as simple aggregates with much better performance. There is no single solution here; but when you’re encountering performance issues, do consider how the calculation can be resolved in the source database or DSV and prepopulated rather than evaluated at query time.

For example, instead of writing expressions like Sum(Customer.City.Members, cint(Customer.City.Currentmember.properties(“Population”))), consider defining a separate measure group on the City table, with a sum measure on the Population column.

As a second example, you can compute the product of revenue \* Products Sold at leaves and aggregate with calculations. Computing this result in the source database or in the DSV will result in superior performance.

#### NON\_EMTPY\_BEHAVIOR

In some situations, it is expensive to compute the result of an expression, even though we know it will be null beforehand based on the value of some indicator tuple. The NONEMPTY\_BEHAVIOR property was sometimes helpful for these kinds of calculations. When this property evaluated to null, the expression was guaranteed to be null and (most of the time) vice versa.

This property oftentimes resulted in substantial performance improvements in past releases. In SQL Server 2008, the property is oftentimes ignored (because the engine automatically deals with nonempty cells in many cases) and can sometimes result in degraded performance. Eliminate it from the MDX script and add it back after performance testing demonstrates improvement.

For assignments, the property is used as follows.

this = <e1>;

Non\_Empty\_Behavior(this) = <e2>;

For calculated members in the MDX script, the property is used this way.

create member currentcube.measures.x as <e1>, non\_empty\_behavior = <e2>

In SQL Server 2005 Analysis Services, there were complex rules on how the property could be defined, when the engine used it or ignored it, and how the engine would use it. In SQL Server 2008 Analysis Services, the behavior of this property has changed:

* It remains a guarantee that when NON\_EMPTY\_BEHAVIOR is null that the expression must also be null. (If this is not true, incorrect query results can still be returned.)
* However, the reverse is not necessarily true; that is, the NON\_EMPTY\_BEHAVIOR expression can return non null when the original expression is null.
* The engine will more often than not ignore this property and deduce the nonempty behavior of the expression on its own.

If the property is defined and is applied by the engine, it is semantically equivalent (not performance equivalent, however) to the following expression.

this = <e1> \* iif(isempty(<e2>), null, 1)

The NON\_EMPTY\_BEHAVIOR property is used if <e2> is sparse and <e1> is dense or <e1> is evaluated in the naïve cell-by-cell mode. If these conditions are not met and both <e1> and <e2> are sparse (i.e., <e2> is much sparser than <e1>), improved performance might be achieved by forcing the behavior as follows.

this = iif(isempty(<e2>), null, <e1>);

The NON\_EMPTY\_BEHAVIOR property can be expressed as a simple tuple expression including simple member navigation functions such as .prevmember or .parent or an enumerated set. An enumerated set is equivalent to NON\_EMPTY\_BEHAVIOR of the resultant sum.

## Cache Warming

During querying, memory is primarily used to store cached results in the storage engine and query processor caches. To optimize the benefits of caching, you can often increase query responsiveness by preloading data into one or both of these caches. This can be done by either pre-executing one or more queries or using the create cache statement. This process is called [*cache warming*](http://sqlcat.com/technicalnotes/archive/2007/09/11/how-to-warm-up-the-analysis-services-data-cache-using-create-cache-statement.aspx).

The two mechanisms are similar although the create cache statement has the advantage of not returning cell values and generally executes faster because the query processor is bypassed.

Discovering what needs to be cached can be difficult. One approach is to run a trace during query execution and examining subcube events. Finding many subcube requests to the same grain may indicate that the query processor is making many requests for slightly different data, resulting in the storage engine making many small but time-consuming I/O requests where it could more efficiently retrieve the data *en masse* and then return results from cache.

To pre-execute queries, you can create an application that executes a set of generalized queries to simulate typical user activity in order to expedite the process of populating the cache. For example, if you determine that users are querying by month and by product, you can create a set of queries that request data by product and by month. If you run this query whenever you start Analysis Services or process the measure group or one of its partitions, this will preload the query results cache with data used to resolve these queries before users submit these types of query. This technique substantially improves Analysis Services response times to user queries that were anticipated by this set of queries.

To determine a set of generalized queries, you can use the Analysis Services query log to determine the dimension attributes typically queried by user queries. You can use an application, such as a Microsoft Excel macro, or a script file to warm the cache whenever you have performed an operation that flushes the query results cache. For example, this application could be executed automatically at the end of the cube processing step.

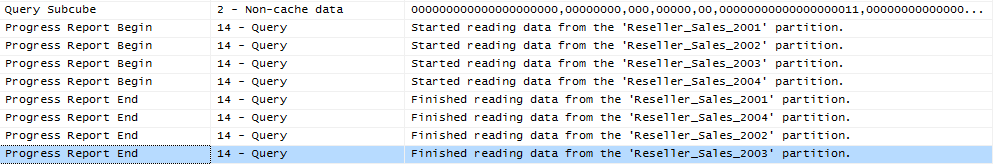
When testing the effectiveness of different cache-warming queries, you should empty the query results cache between each test to ensure the validity of your testing.

Note that the cached results can be pushed out by other query results. It may be necessary to refresh the cache results according to some schedule. Also, limit cache warming to what can fit in memory leaving enough for other queries to be cached.

**Aggressive Data Scanning**

It is possible that in the evaluation of an expression more data is requested than required to determine the result.

If you suspect more data is being retrieved than is required, you can use SQL Server Profiler to diagnose how a query into subcube query events and partition scans. For subcube scans, check the verbose subcube event and whether more members than required are retrieved from the storage engine. For small cubes, this likely isn’t a problem. For larger cubes with multiple partitions, it can greatly reduce query performance. The following figure demonstrates how a single query subcube event results in partition scans.

There are two potential solutions to this. If a calculation expression contains an arbitrary shape (this is defined in the section on the query processor cache), the query processor may not be able to determine that the data is limited to a single partition and request data from all partitions. Try to eliminate the arbitrary shape.

Other times, the query processor is simply overly aggressive in asking for data. For small cubes, this doesn’t matter, but for very large cubes, it does. If you observe this behavior, contact Microsoft Customer Service and Support for further advice.

## Improving Multiple-User Performance

In many cases, poor multiple-user performance can be traced to poor single-user performance. But this isn’t always true. In some cases, Analysis Services does not exploit all the resources on the computer when scaling up the number of users. There are a few options to improve performance.

### Increasing Query Parallelism

During querying, to manage client connections, Analysis Services uses a listener thread to broker requests and create new server connections as needed. To satisfy query requests, the listener thread manages worker threads in the querying thread pool and the processing thread pool, assigns worker threads to specific requests, initiates new worker threads if there are not enough active worker threads in a given pool, and terminates idle worker threads as needed.

To satisfy a query request, the thread pools are used as follows:

* Worker threads from the query pool check the data and calculation caches respectively for any data and/or calculations pertinent to a client request.
* If necessary, worker threads from the processing pool are allocated to retrieve data from disk.
* After data is retrieved, worker threads from the querying pool store the results in the query cache to resolve future queries.
* Worker threads from the querying pool perform necessary calculations and use a calculation cache to store calculation results.

The more threads that are available to satisfy queries, the more queries that you can execute in parallel. This is especially important in scenarios where you have a large number of users issuing queries.

**Threadpool\Query\MaxThreads** determines the maximum number of worker threads maintained in the querying thread pool. The default value of this property is either 10 or 2x the number of cores (this is different from SQL Server 2005, so check this value for upgraded instances). Increasing **Threadpool\Query\MaxThreads** will not significantly increase the performance of a given query. Rather, the benefit of increasing this property is that you can increase the number of queries that can be serviced concurrently.

Because querying also involves retrieving data from partitions, you must also consider the maximum threads available in the processing pool as specified by the **Threadpool\Process\MaxThreads** property. By default, this property has a value of 64 in SQL Server 2005. This changed to either 64 or 10 times the number of cores, whichever is greater, in SQL Server 2008 (and this is the currently recommended value). As you consider the scenarios for changing the **Threadpool\Process\MaxThreads** property, remember that changing this setting impacts the processing thread pool for both querying and processing. For more information about how this property specifically impacts processing operations, see Adjusting Thread Settings.

While modifying the **Threadpool\Process\MaxThreads** and **Threadpool\Query\MaxThreads** properties can increase parallelism during querying, you must also take into account the additional impact of **CoordinatorExecutionMode**. Consider the following example. If you have a four-processor server and you accept the default **CoordinatorExecutionMode** setting of -4, a total of 16 jobs can be executed at one time across all server operations. So if ten queries are executed in parallel and require a total of 20 jobs, only 16 jobs can launch at a given time (assuming that no processing operations are being performed at that time). When the job threshold has been reached, subsequent jobs wait in a queue until a new job can be created. Therefore, if the number of jobs is the bottleneck to the operation, increasing the thread counts may not necessarily improve overall performance.

In practical terms, the balancing of jobs and threads can be tricky. If you want to increase parallelism, it is important to assess your greatest bottleneck to parallelism, such as the number of concurrent jobs and/or the number of concurrent threads, or both. To help you determine this, it is helpful to monitor the following performance counters:

* **Threads\Query pool job queue length**—The number of jobs in the queue of the query thread pool. A nonzero value means that the number of query jobs has exceeded the number of available query threads. In this scenario, you may consider increasing the number of query threads. However, if CPU utilization is already very high, increasing the number of threads will only add to context switches and degrade performance.
* **Threads\Query pool busy threads**—The number of busy threads in the query thread pool.
* **Threads\Query pool idle threads**—The number of idle threads in the query thread pool.

More information is available at <http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ssasqptb.mspx>.

### Memory Heap Type

Multiple-user throughput can be improved by using the Windows heap instead of the Analysis Services heap. This increase in throughput can come at a small but measurable cost to single-user queries, but multiple-user throughput has been measured to improve by up to 100% (that is, twice as many queries being managed in the same amount of time). The significant benefits of multiple-user throughput generally outweigh the single-user performance cost. To use the NTLFH heap manager instead of the OLAP heap manager, change the following parameters.

| **Setting** | **Default** | **Multiple-user** |
| --- | --- | --- |
| MemoryHeapType | 1 | 2 |
| HeapTypeForObjects | 1 | 0 |

### Blocking Long-Running Queries

In multiple-user scenarios, long-running queries can starve other queries – even shorter-running queries – by consuming all available threads, and they can block execution of other queries until the longer-running query has completed. You can reduce the aggressiveness of how each coordinator job consumes queries by changing how each segment job is queued up. As discussed in Data Retrieval, each segment job is immediately queued up before the prior segment job begins scanning data. You can change this behavior to serialize segment jobs by using the following settings.

| **Setting** | **Default** | **Multiple-user nonblocking settings** |
| --- | --- | --- |
| CoordinatorQueryBalancingFactor | -1 | 1 |
| CoordinatorQueryBoostPriorityLevel | 3 | 0 |

But be careful – although these settings reduce or stop the blocking of shorter-running queries by longer-running ones, it lessens the overall throughput. If you change these settings and still see blocking queries, contact Microsoft Customer Service and Support.

### Network Load Balancing and Read-Only Databases

Although they are beyond the scope of this document, fundamental design changes can be brought to bear to address query issues and are briefly described here.

#### Network Load Balancing

If your performance bottleneck is processor utilization on a single system as a result of a multiple-user query workload, you can increase query performance by using a cluster of Analysis Services servers to service query requests. Requests can be load balanced across two Analysis Services servers, or across a larger number of Analysis Services servers to support a large number of concurrent users (this is called a *server farm*). Load-balancing clusters generally scale linearly. Both Microsoft and third-party vendors provide cluster solutions. The Microsoft load-balancing solution is Network Load Balancing (NLB), which is a feature of the Windows Server® operating system. With NLB, you can create an NLB cluster of Analysis Services servers running in multiple host mode. When an NLB cluster of Analysis Services servers is running in multiple host mode, incoming requests are load balanced among the Analysis Services servers. When you use a load-balancing cluster, be aware that the data caches on each of the servers in the load-balancing cluster will be different, resulting in differences in query response times from query to query by the same client.

A load-balancing cluster can also be used to ensure availability in the event that a single Analysis Services server fails. An additional option for increasing performance with a load-balancing cluster is to distribute processing tasks to an offline server. When new data has been processed on the offline server, you can update the Analysis Services servers in the load-balancing cluster by using Analysis Services database synchronization.

If your users submit a lot of queries that require fact data scans, a load-balancing cluster may be a good solution. For example, queries that may require a large number of fact data scans include wide queries (such as top count or medians), and random queries against very complex cubes where the probability of hitting an aggregation is very low.

However, a load-balancing cluster is generally not needed to increase Analysis Services performance if aggregations are being used to resolve most queries. In other words, concentrate on good aggregation and partitioning design first. In addition, a load-balancing cluster does not solve your performance problem if processing is the bottleneck or if you are trying to improve an individual query from a single user. Note that one restriction to using a load-balancing cluster is the inability to use writeback, because there is no single server to which to write back the data.

#### Read-Only Databases

New in SQL Server 2008, a database can be marked as read-only and used by multiple instances of Analysis Services; that is, multiple instances of Analysis Services can share a single data directory (typically located on a SAN). This option should be considered if multiple-user workload is light on storage engine requirements but heavy on query processor. While Analysis Services supports multiple instances pointing to the same data folder, it is up to the application to manage user sessions across these instances.

# Understanding and Measuring Processing

In the following sections we will provide guidance on tuning processing of cubes. Processing is the operation that loads data from one or more data sources into one or more Analysis Services objects. While OLAP systems are not generally judged by how fast they process data, processing performance impacts how quickly new data is available for querying. While every application has different data refresh requirements, ranging from monthly updates to near real-time data refreshes, the faster the processing performance, the sooner users can query refreshed data.

Analysis Services provides several processing commands, allowing granular control over the data loading and refresh frequency of cubes.

## Processing Job Overview

To manage processing operations, Analysis Services uses centrally controlled jobs. A processing job is a generic unit of work generated by a processing request.

From an architectural perspective, a job can be broken down into parent jobs and child jobs. For a given object, you can have multiple levels of nested jobs depending on where the object is located in the OLAP database hierarchy. The number and type of parent and child jobs depend on 1) the object that you are processing, such as a dimension, cube, measure group, or partition, and 2) the processing operation that you are requesting, such as **ProcessFull**, **ProcessUpdate**, or **ProcessIndexes**.

For example, when you issue a **ProcessFull** operation for a measure group, a parent job is created for the measure group with child jobs created for each partition. For each partition, a series of child jobs are spawned to carry out the **ProcessFull** operation of the fact data and aggregations. In addition, Analysis Services implements dependencies between jobs. For example, cube jobs are dependent on dimension jobs.

The most significant opportunities to tune performance involve the processing jobs for the core processing objects: dimensions and partitions. Each of these has its own section in this guide.

Additional background information on processing can be found in the technical note [Analysis Services 2005 Processing Architecture](http://msdn.microsoft.com/en-us/library/ms345142(SQL.90).aspx).

## Baselining Processing

To quantify the effects of your tuning and diagnose problems, you should first create a baseline. The baseline allows you to analyze root causes and to target optimization effort.

This section describes how to set up the baseline.

### Performance Monitor Trace

Windows Performance counters are the bread and butter of performance tuning Analysis Services. Use the tool **perfmon** to set up a trace with these counters:

* **MSOLAP: Processing** 
  + **Rows read/sec**
* **MSOLAP: Proc Aggregations** 
  + **Temp File Bytes Writes/sec**
  + **Rows created/Sec**
  + **Current Partitions**
* **MSOLAP: Threads**
  + **Processing pool idle threads**
  + **Processing pool job queue length**
  + **Processing pool busy threads**
* **MSSQL: Memory Manager**
  + **Total Server Memory**
  + **Target Server Memory**
* **Process** 
  + **Virtual Bytes – msmdsrv.exe**
  + **Working Set – msmdsrv.exe**
  + **Private Bytes – msmdsrv.exe**
  + **% Processor Time – msmdsrv.exe and sqlservr.exe**
* **Logical Disk:** 
  + **Avg. Disk sec/Transfer – All Instances**
* **Processor:** 
  + **% Processor Time – Total**
* **System:**
  + **Context Switches / sec**

Configure the trace to save data to a file. Measuring every 15 seconds will be sufficient for tuning processing.

As you tune processing, you should measure these counters again after each change to see if you are getting closer to your performance goal. Also note the total time used by processing. The use and interpretation of the individual counters will be explained in the sections below.

### Profiler Trace

To optimize the SQL queries that form part of processing, you should trace the relational database too. If the relational database is SQL Server, you use SQL Server Profiler for this. If you are not using SQL Server, consult your database vendor for help on tuning the database. In the following we will assume that you use SQL Server as the relational foundation for Analysis Services.

In your SQL Server Profiler trace you should also capture the events:

* **Performance**/**Showplan XML Statistics Profile**
* **TSQL/SQL:BatchCompleted**

Include these event columns:

* **TextData**
* **Reads**
* **DatabaseName**
* **SPID**
* **Duration**

You can use the **Tuning** template and just add the **Reads** column and **Showplan XML Statistics Profiles**. Like the **perfmon** trace, configure the trace to save to a file for later analysis.

Configure your SQL Server Profiler trace to log to a table instead of a file. This makes it easier to correlate the traces later.

The performance data gathered by these traces will be used in the following section to help you tune processing.

## Determining Where You Spend Processing Time

To properly target the tuning of processing, you should first determine where you are spending your time: partition processing or dimension processing. Because dimensions are processed before partitions, you can easily measure how much time you spend on dimensions.

For partition processing, you should distinguish between **ProcessData** and **ProcessIndex** – the tuning techniques for each are very different. If you follow our recommended best practice of doing **ProcessData** followed by **ProcessIndex** instead of **ProcessFull**, the time spent in each should be easy to read.

If you use **ProcessFull** instead of splitting into **ProcessData** and **ProcessIndex**, you can get an idea of when each phase ends by observing the following **Perfmon** counters:

* During **ProcessData** the counter **MSOLAP:Processing – Rows read/Sec** is greater than zero.
* During **ProcessIndex** the counter **MSOLAP:Proc Aggregations – Row created/Sec** is greater than zero.

**ProcessData** can be further split into the time spent by the SQL Server process and the time spent by the Analysis Services process. You can use the **Process** counters collected to see where most of the CPU time is spent.

# Enhancing Dimension Processing Performance

The performance goal of dimension processing is to refresh dimension data in an efficient manner that does not negatively impact the query performance of dependent partitions. The following techniques for accomplishing this goal are discussed in this section:

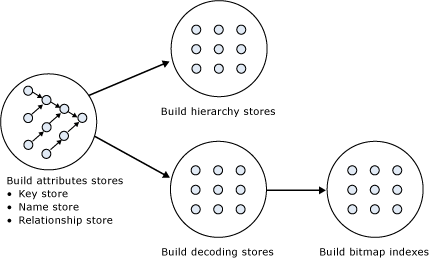
* Optimizing SQL source queries.
* Reducing attribute overhead.

This section also includes information about dimension processing architecture.

## Understanding Dimension Processing Architecture

During the processing of MOLAP dimensions, jobs are used to extract, index, and persist data in a series of dimension stores.

To create these dimension stores, the storage engine uses the series of jobs displayed in Figure 19.



**Figure 19 Dimension processing jobs**

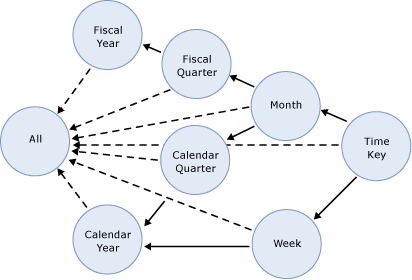
**Build Attribute Stores**

For each attribute in a dimension, a job is instantiated to extract and persist the attribute members into an attribute store. The attribute store consists of the key store, name store, and relationship store.

Because the relationship stores contain information about dependent attributes, an ordering of the processing jobs is required. To provide the correct workflow, the storage engine analyzes the dependencies between attributes, and then it creates an execution tree with the correct ordering. The execution tree is then used to determine the best parallel execution of the dimension processing.

Figure 20 displays an example execution tree for a Time dimension. The solid arrows represent the attribute relationships in the dimension. The dashed arrows represent the implicit relationship of each attribute to the All attribute.

**Note:** The dimension has been configured using cascading attribute relationships, which is a best practice for all dimension designs.



**Figure 20 Execution tree example**

In this example, the All attribute proceeds first, given that it has no dependencies to another attribute, followed by the Fiscal Year and Calendar Year attributes, which can be processed in parallel. The other attributes proceed according to the dependencies in the execution tree, with the primary key attribute always being processed last, since it always has at least one attribute relationship, except when it is the only attribute in the dimension.

The time taken to process an attribute is generally dependent on 1) the number of members and 2) the number of attribute relationships. While you cannot control the number of members for a given attribute, you can improve processing performance by using cascading attribute relationships. This is especially critical for the key attribute, since it has the most members and all other jobs (hierarchy, decoding, bitmap indexes) are waiting for it to complete. Attribute relationships will lower the memory requirement during processing. When an attribute is processed, all dependent attributes must be kept in memory. If you had no attribute relationships, all attributes would have to be kept in memory while the key attribute was being processed. This may cause out-of-memory conditions.

For more information about the importance of using cascading attribute relationships, see Identifying Attribute Relationships.

**Build Decoding Stores**

Decoding stores are used extensively by the storage engine. During querying, they are used to retrieve data from the dimension. During processing, they are used to build the dimension’s bitmap indexes.

**Build Hierarchy Stores**

A *hierarchy store* is a persistent representation of the tree structure. For each natural hierarchy in the dimension, a job is instantiated to create the hierarchy stores.

**Build Bitmap Indexes**

To efficiently locate attribute data in the relationship store at querying time, the storage engine creates bitmap indexes at processing time. For attributes with a very large number of members, the bitmap indexes can take some time to process. In most scenarios, the bitmap indexes provide significant querying benefits; however, when you have high-cardinality attributes, the querying benefit that the bitmap index provides may not outweigh the processing cost of creating the bitmap index.

## Dimension-Processing Commands

When you need to perform a process operation on a dimension, you issue dimension processing commands. Each processing command creates one or more jobs to perform the necessary operations.

From a performance perspective, the following dimension processing commands are the most important:

* **ProcessFull**
* **ProcessData**
* **ProcessIndexes**
* **ProcessUpdate**
* **ProcessAdd**

A **ProcessFull** command discards all storage contents of the dimension and rebuilds them. Behind the scenes, **ProcessFull** executes all dimension processing jobs and performs an implicit **ProcessClear** on all dependent partitions. This means that whenever you perform a **ProcessFull** operation of a dimension, you need to perform a **ProcessFull** operation on dependent partitions to bring the cube back online.

**ProcessData** discards all storage contents of the dimension and rebuilds only the attribute and hierarchy stores and also clears partitions. **ProcessData** is the first component executed by a **ProcessFull** operation.

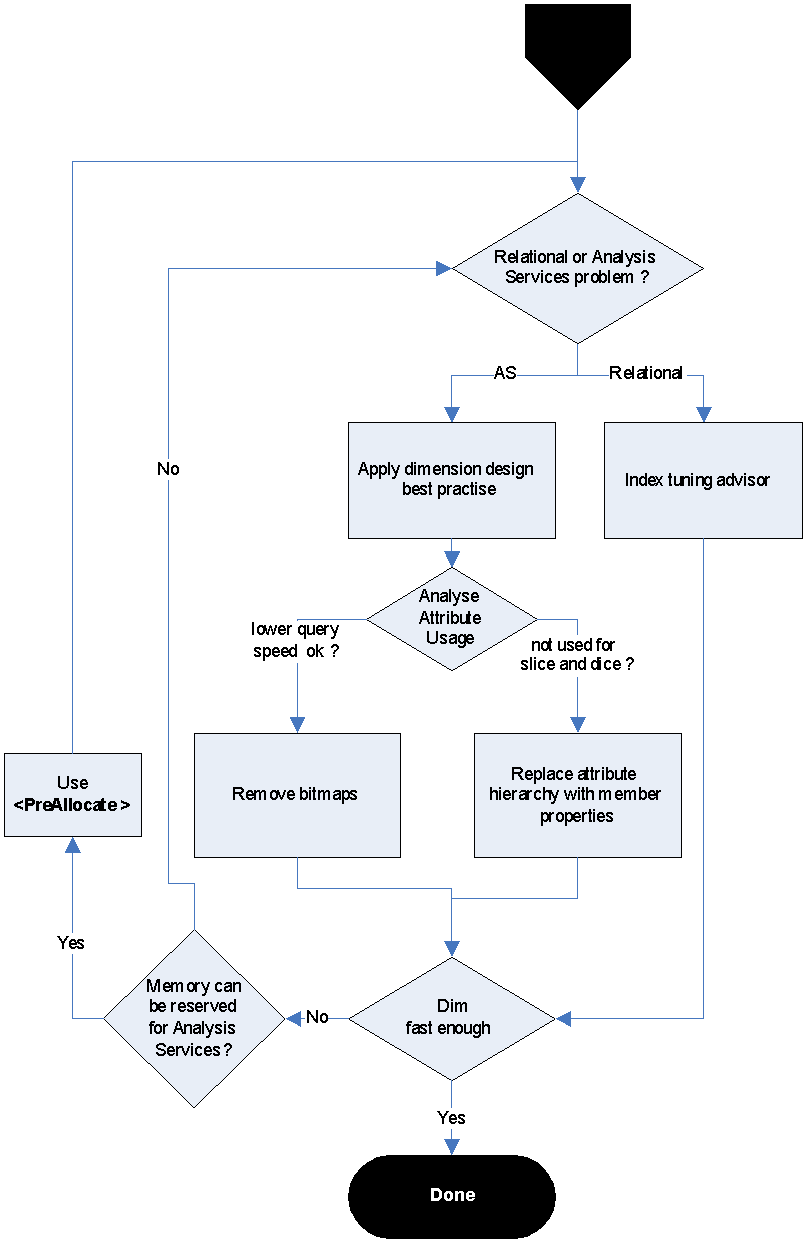
**ProcessIndexes** requires that a dimension already has attribute and hierarchy stores built it preserves the data in these stores and then rebuilds the bitmap indexes. **ProcessIndexes** is the second component of the **ProcessFull** operation.

Unlike **ProcessFull**, **ProcessUpdate** does not discard the dimension storage contents. Instead, it applies updates intelligently in order to preserve dependent partitions. More specifically, **ProcessUpdate** sends SQL queries to read the entire dimension table and then applies changes to the dimension stores. A **ProcessUpdate** can handle inserts, updates, and deletions, depending on the type of attribute relationships (rigid vs. flexible) in the dimension. Note that **ProcessUpdate** will drop invalid aggregations and indexes, requiring you to take action to rebuild the aggregations in order to maintain query performance. However, flexible aggregations are only dropped if a change is detected.

**ProcessAdd** optimizes **ProcessUpdate** in scenarios where you only need to insert new members. **ProcessAdd** does not delete or update existing members. The performance benefit of **ProcessAdd** is that you can use a different source table or data source view named query that restrict the rows of the source dimension table to only return the new rows. This eliminates the need to read all of the source data. In addition, **ProcessAdd** also retains all indexes and aggregations (flexible and rigid).

**Note:** **ProcessAdd** is only available as an XMLA command.

## Dimension Processing Tuning Flow Chart



For information about SQL Server wait statistics and how to track them, see [**sys.dm\_os\_wait\_stats**](http://msdn.microsoft.com/en-us/library/ms179984.aspx) in SQL Server Books Online.

## Dimension Processing Performance Best Practices

There are some general, good design practices that are simple to implement and which provide some quick wins for performance of dimension. You should seek to incorporate these in your best practices when designing dimensions.

In SQL Server 2008 Analysis Services, the Analysis Management Objects (AMO) warnings are provided by Business Intelligence Development Studio to assist you with designing best practices.

### Use SQL Views to Implement Query Binding for Dimensions

While query binding for dimensions does not exist in SQL Server 2008 Analysis Services, you can implement it by using a view (instead of tables) for your underlying dimension data source. That way, you can use hints, indexed views, or other relational database tuning techniques to optimize the SQL statement that accesses the dimension tables through your view.

It is generally a good idea to build your Unified Dimensional Model (UDM) on top of database views. Not only can you apply relational tuning, you can also use the NOLOCK hint in the view definition. This hint removes locking overhead from the database, which can benefit performance even further.

Views provide easy of debugging. You can issue SQL queries directly on views to compare the relational data with the cube. Hence, views provide a good way to encapsulate business logic that you would normally implement as query binding in the UDM. While the UDM syntax is similar to the SQL view syntax, you cannot issue SQL statements against the UDM.

### Optimize Attribute Processing Across Multiple Data Sources

When a dimension comes from multiple data sources, using cascading attribute relationships allows the system to segment attributes during processing according to data source. If an attribute’s key, name, and attribute relationships come from the same database, the system can optimize the SQL query for that attribute by querying only one database. Without cascading attribute relationships, the SQL Server OPENROWSET function, which provides a method for accessing data from multiple sources, is used to merge the data streams. In this situation, the processing for the key attribute is extremely slow, because it must access multiple OPENROWSET derived tables.

If you have the option, consider performing ETL to bring all data needed for the dimension into the same SQL Server database. This allows you to utilize the Relational Engine to tune the query.

### Reduce Attribute Overhead

Every attribute that you include in a dimension impacts the cube size, the dimension size, the aggregation design, and processing performance. Whenever you identify an attribute that will not be used by end users, delete the attribute entirely from your dimension. After you have removed extraneous attributes, you can apply a series of techniques to optimize the processing of remaining attributes.

### Use the KeyColumns, ValueColumn, and NameColumn Properties Effectively

When you add a new attribute to a dimension, three properties are used to define the attribute. The **KeyColumns** property specifies one or more source fields that uniquely identify each instance of the attribute.

The **NameColumn** property specifies the source field that will be displayed to end users. If you do not specify a value for the **NameColumn** property, it is automatically set to the value of the **KeyColumns** property.

**ValueColumn** allows you to carry further information about the attribute – typically used for calculations. Unlike member properties, this property of an attribute is strongly typed – providing increased performance when it is used in calculations. The contents of this property can be accessed through the **MemberValue** MDX function.

Using **ValueColumn** and **NameColumn** eliminates the need for extraneous attributes. This reduces the total number of attributes in your design, making it more efficient.

Analysis Services provides the ability to source the **KeyColumns**, **ValueColumn**, and **NameColumn** properties from different source columns. This is useful when you have a single entity like a product that is identified by two different attributes: a surrogate key and a descriptive product name. When users want to slice data by products, they may find that the surrogate key lacks business relevance and will choose to use the product name instead.

It is a best practice to assign a numeric source field to the **KeyColumns** property rather than a string property. Not only does this reduce processing time, in also reduces the size of the dimension. This is especially true for attributes that have a large number of members, i.e., greater than one million members.

### Remove Bitmap Indexes

During processing of the primary key attribute, bitmap indexes are created for every related attribute. Building the bitmap indexes for the primary key can take time if it has one or more related attributes with high cardinality. At query time, the bitmap indexes for these attributes are not useful in speeding up retrieval, since the storage engine still must sift through a large number of distinct values. This may have a negative impact on query response times.

For example, the primary key of the customer dimension uniquely identifies each customer by account number; however, users also want to slice and dice data by the customer’s social security number. Each customer account number has a one-to-one relationship with a customer social security number. To avoid spending time building unnecessary bitmap indexes for the social security number attribute, it is possible to disable its bitmap indexes by setting the **AttributeHierarchyOptimizedState** property to **Not Optimized**.

### Turn Off the Attribute Hierarchy and Use Member Properties

As an alternative to attribute hierarchies, member properties provide a different mechanism to expose dimension information. For a given attribute, member properties are automatically created for every attribute relationship. For the primary key attribute, this means that every attribute that is directly related to the primary key is available as a member property of the primary key attribute.

If you only want to access an attribute as member property, after you verify that the correct relationship is in place, you can disable the attribute’s hierarchy by setting the **AttributeHierarchyEnabled** property to **False**. From a processing perspective, disabling the attribute hierarchy can improve performance and decrease cube size because the attribute will no longer be indexed or aggregated. This can be especially useful for high-cardinality attributes that have a one-to-one relationship with the primary key. High-cardinality attributes such as phone numbers and addresses typically do not require slice-and-dice analysis. By disabling the hierarchies for these attributes and accessing them via member properties, you can save processing time and reduce cube size.

Deciding whether to disable the attribute’s hierarchy requires that you consider both the querying and processing impacts of using member properties. Member properties cannot be placed on a query axis in Business Intelligence Design Studio in the same manner as attribute hierarchies and user hierarchies. To query a member property, you must query the properties of the attribute that contains the member property.

For example, if you require the work phone number for a customer, you must query the properties of customer and then request the phone number property. As a convenience, most front-end tools easily display member properties in their user interfaces.

In general, querying member properties can be slower than querying attribute hierarchies, because member properties are not indexed and do not participate in aggregations. The actual impact to query performance depends on how you are going to use the attribute.

For example, if your users want to slice and dice data by both account number and account description, from a querying perspective you may be better off having the attribute hierarchies in place and removing the bitmap indexes if processing performance is an issue.

## Tuning the Relational Dimension Processing Query

Unlike fact partitions, which only send one query to the server, dimension process operations will send multiple queries. Dimensions tend to be small, complex tables with very few changes, compared to facts. Tables with such characteristics can often be heavily indexed with little insert/update performance overhead to the system. You can use this to your advantage during processing and be wasteful with the relational indexes.

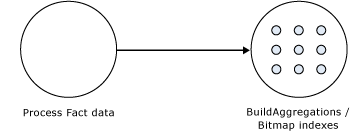
The easiest way to tune the relational queries used for dimension processing is to use the Database Engine Tuning Advisor on a profiler trace of the dimension processing. For the small dimension tables, chances are that you can get away with adding every suggested index. For the larger tables, target the indexes towards the longest-running queries.

# Enhancing Partition Processing Performance

The performance goal of partition processing is to refresh fact data and aggregations in an efficient manner that satisfies your overall data refresh requirements. The following techniques for accomplishing this goal are discussed in this section: optimizing SQL source queries, using multiple partitions, tuning I/O, optimizing networking speeds, and tuning thread and concurrency settings.

## Understanding the Partition Processing Architecture

During partition processing, source data is extracted and stored on disk using the series of jobs displayed In Figure 21.



**Figure 21 Partition processing jobs**

**Process Fact Data**

Fact data is processed using three concurrent threads that perform the following tasks:

* Send SQL statements to extract data from data sources.
* Look up dimension keys in dimension stores and populate the processing buffer.
* When the processing buffer is full, write out the buffer to disk.

**Build Aggregations and Bitmap Indexes**

Aggregations are built in memory during processing. While too few aggregations may have little impact on query performance, excessive aggregations can increase processing time without much added value on query performance.

If aggregations do not fit in memory, chunks are written to temp files and merged at the end of the process. Bitmap indexes are also built during this phase and written to disk on a segment-by-segment basis.

## Partition-Processing Commands

When you need to perform a process operation on a partition, you issue partition processing commands. Each processing command creates one or more jobs to perform the necessary operations.

The following partition processing commands are available:

* **ProcessFull**
* **ProcessData**
* **ProcessIndexes**
* **ProcessAdd**
* **ProcessClear**
* **ProcessClearIndex**

**ProcessFull** discards the storage contents of the partition and rebuilds them. Behind the scenes, a**ProcessFull** executes **ProcessData** and **ProcessIndexes** jobs.

**ProcessData** discards the storage contents of the object and rebuilds only the fact data.

**ProcessIndexes** requires a partition to have built its data already. **ProcessIndexes** preserves the data created during **ProcessData** and creates new aggregations and bitmap indexes based on it.

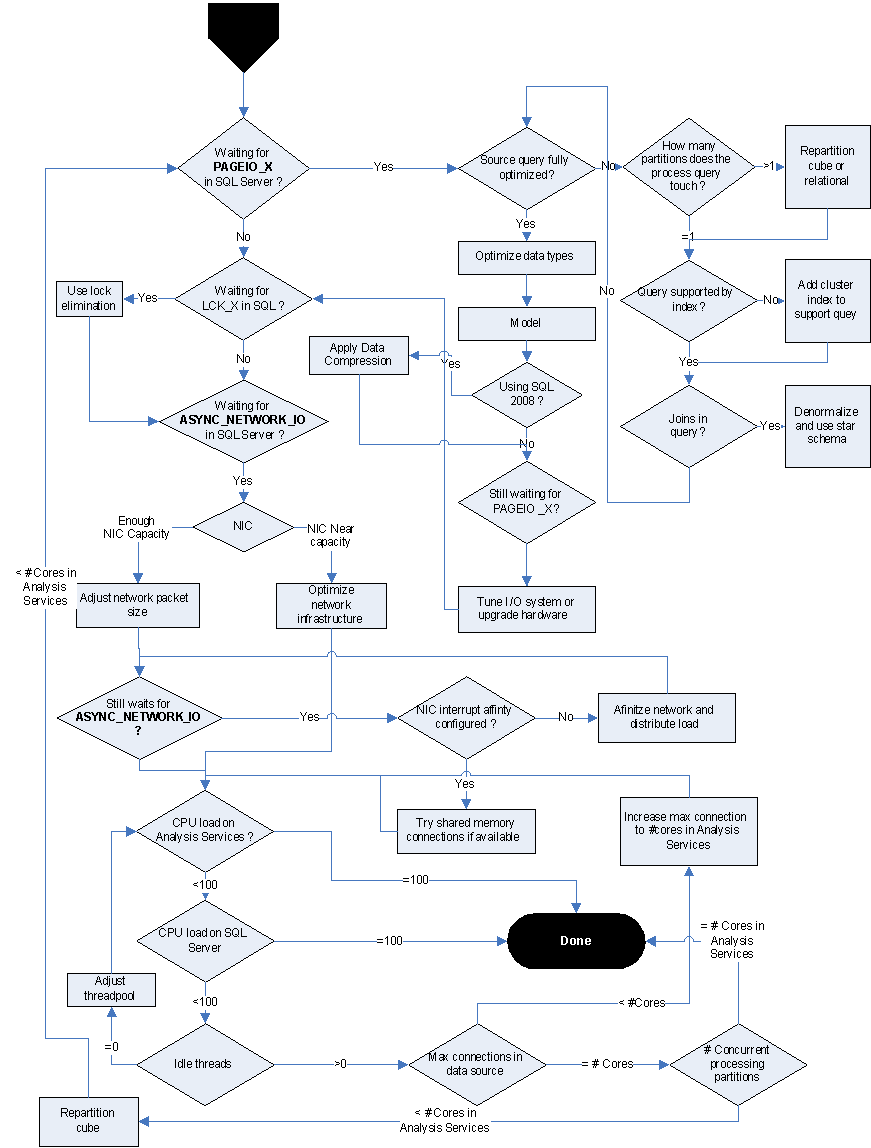
**ProcessAdd** internally creates a temporary partition, processes it with the target fact data, and then merges it with the existing partition. Note that **ProcessAdd** is the name of the XMLA command, in Business Intelligence Development Studio and SQL Server Management Studio this is exposed as **ProcessIncremental.**

**ProcessClear** removes all data from the partition.Note the **ProcessClear** is the name of the XMLA command. In Business Intelligence Development Studio and SQL Server Management Studio, it is exposed as **UnProcess.**

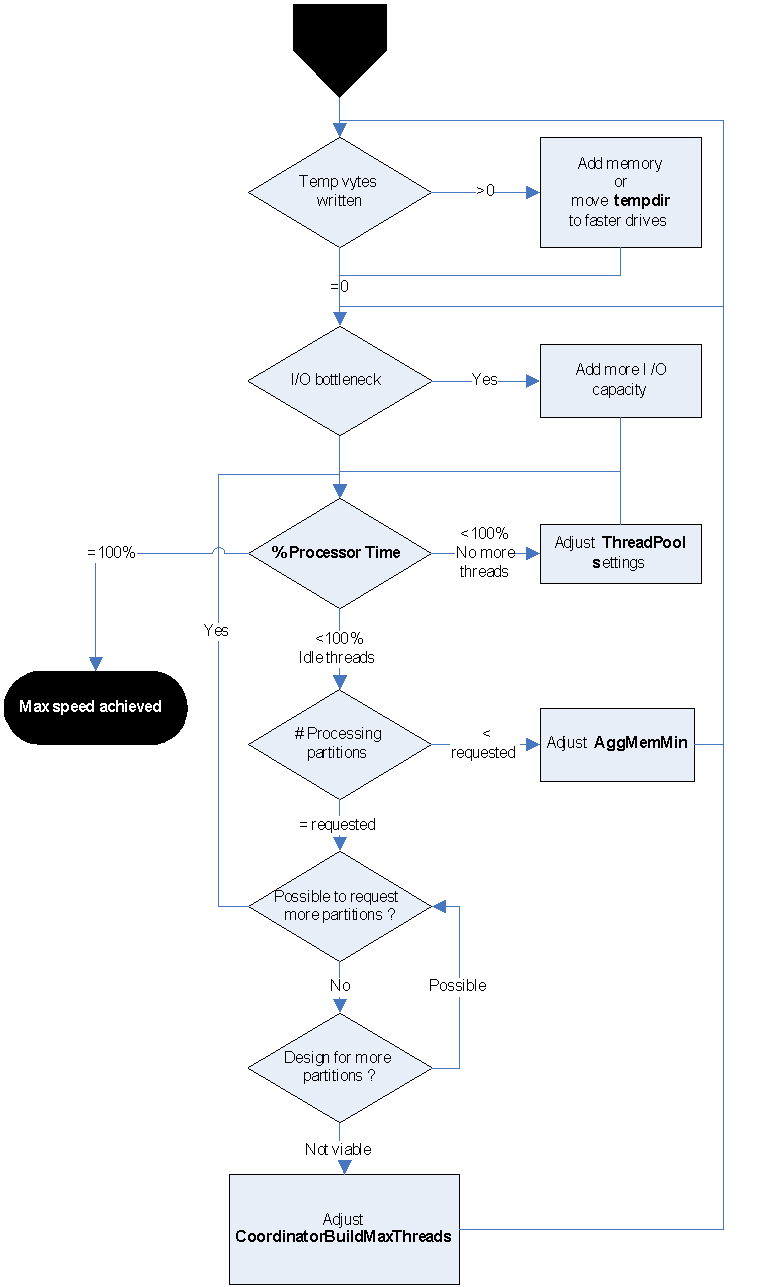
**ProcessClearIndexes** removes all indexes and aggregates from the partition. This brings the partitions in the same state as if **ProcessClear** followed by **ProcessData** had just been run.Note that **ProcessClearIndexes** is the name of the XMLA command. This command is not available in Business Intelligence Development Studio and SQL Server Management Studio.

## Partition Processing Tuning Flow Chart

The following flowchart describes the tuning process of **ProcessData**.



For **ProcessIndexes**, the flowchart below applies.



## Partition Processing Performance Best Practices

When designing your fact tables, use the guidance in the following technical notes:

* [Top 10 Best Practices for Building a Large Scale Relational Data Warehouse](http://sqlcat.com/top10lists/archive/2008/02/06/top-10-best-practices-for-building-a-large-scale-relational-data-warehouse.aspx)
* [Analysis Services Processing Best Practices](http://sqlcat.com/whitepapers/archive/2007/11/15/analysis-services-processing-best-practices.aspx)

### Optimizing Data Inserts, Updates, and Deletes

This section provides guidance on how to efficiently refresh partition data to handle inserts, updates, and deletes.

**Inserts**

If you have a browsable, processed cube and you need to add new data to an existing measure group partition, you can apply one of the following techniques:

* **ProcessFull**—Perform a **ProcessFull** operation for the existing partition. During the **ProcessFull** operation, the cube remains available for browsing with the existing data while a separate set of data files are created to contain the new data. When the processing is complete, the new partition data is available for browsing. Note that **ProcessFull** is technically not necessary, given that you are only doing inserts. To optimize processing for insert operations, you can use **ProcessAdd**.
* **ProcessAdd**—Use this operation to append data to the existing partition files. If you frequently perform **ProcessAdd**, it is advised that you periodically perform **ProcessFull** in order to rebuild and recompress the partition data files. **ProcessAdd** internally creates a temporary partition and merges it. This results in data fragmentation over time and the need to periodically perform **ProcessFull**.

If your measure group contains multiple partitions, as described in the previous section, a more effective approach is to create a new partition that contains the new data and then perform **ProcessFull** on that partition. This technique allows you to add new data without impacting the existing partitions. When the new partition has completed processing, it is available for querying.

**Updates**

When you need to perform data updates, you can perform a **ProcessFull**. Of course it is useful if you can target the updates to a specific partition so you only have to process a single partition. Rather than directly updating fact data, a better practice is to use a *journaling* mechanism to implement data changes. In this scenario, you turn an update into an insertion that corrects that existing data. With this approach, you can simply continue to add new data to the partition by using a **ProcessAdd**. By using journaling, you also have an audit trail of the changes that have been made to the fact table.

**Deletes**

For deletions, multiple partitions provide a great mechanism for you to roll out expired data. Consider the following example. You currently have 13 months of data in a measure group, 1 month per partition. You want to roll out the oldest month from the cube. To do this, you can simply delete the partition without affecting any of the other partitions.

If there are any old dimension members that only appeared in the expired month, you can remove these using a **ProcessUpdate** operation on the dimension (but only if it contains flexible relationships). In order to delete members from the key/granularity attribute of a dimension, you must set the dimension’s **UnknownMember** property to **Hidden**. This is because the server does not know if there is a fact record assigned to the deleted member. With this property set appropriately, the member will be hidden at query time. Another option is to remove the data from the underlying table and perform a **ProcessFull** operation. However, this may take longer than **ProcessUpdate**.

As your dimension grows larger, you may want to perform a **ProcessFull** operation on the dimension to completely remove deleted keys. However, if you do this, all related partitions must also be reprocessed. This may require a large batch window and is not viable for all scenarios.

### Picking Efficient Data Types in Fact Tables

During processing, data has to be moved out of SQL Server and into Analysis Services. The wider your rows are, the more bandwidth must be spent moving the rows.

Some data types are, by the nature of their design, faster to use than others. For fastest performance, consider using only these data types in fact tables.

| Fact column type | Fastest SQL Server data types |
| --- | --- |
| Surrogate keys | **tinyint**, **smallint**, **int**, **bigint** |
| Date key | **int** in the format yyyyMMdd |
| Integer measures | **tinyint**, **smallint**, **int**, **bigint** |
| Numeric measures | **smallmoney**, **money**, **real**, **float**  *(*Note that **decimal** and **vardecimal** require more CPU power to process than **money** and **float** types) |
| Distinct count columns | **tinyint**, **smallint**, **int**, **bigint**  *(*If your count column is **char**, consider either hashing or replacing with surrogate key*)* |

## Tuning the Relational Partition Processing Query

During the **ProcessData** phase, rows are read from a relational source and into Analysis Services. Analysis Services can consume rows at a very high rate during this phase. To achieve these high speeds, you need to tune the relational database to provide a proper throughput.

In the subsection below, we will assume that your relational source is SQL Server. If you are using another relational source, some of the advice still applies – consult your database specialist for platform specific guidance.

Analysis Services will use the partition information to generate the query. Unless you have done any query binding in the UDM, the SELECT statement issues to the relational source is very simple. It consists of:

* A SELECT of the columns required to process. This will be the dimension columns and the measures.
* Optionally, a WHERE criterion if you use partitions. You can control this WHERE criterion by changing the query binding of the partition.

### Getting Rid of Joins

If you are using a database view or a UDM named query as the basis of partitions, you should seek to eliminate joins in this query. You can achieve this by denormalizing the joined columns to the fact table. If you are using a star schema design, you should already have done this.

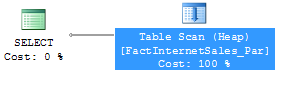
For background on relational star schemas and how to design and denormalize for optimal performance, refer to:

* Ralph Kimball, *The Data Warehouse Toolkit*

### Getting Relational Partitioning Right

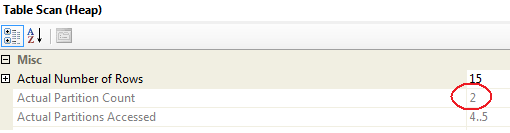
If you use partitioning on the relational side, you should ensure that each cube partition touches at most one relational partition. To check this, use the **XML Showplan** event from your SQL Server Profiler trace.

If you got rid of all joins, your query plan should look something like Figure 22.



**Figure 22 An optimal partition processing query**

Click on the table scan (it may also be a range scan or index seek in your case) and bring up the properties pane.



**Figure 23 Too many partitions accessed**

Both partition 4 and partition 5 are accessed. The value for **Actual Partition Count** should be 1. If this is not the case (as above), you should consider repartitioning the relational source data so that each cube partition touches at most one relational partition.

#### Special Case: Distinct Count

Distinct count measure groups have special requirements for partitioning. Normally, you use time or some other dimension as the partitioning column. However, if you choose to partition a distinct count measure group, you should partition on the value of the distinct count measure column.

Group the distinct count measure column into separate, nonoverlapping intervals. Each interval should contain approximately the same amount of rows from the source. These intervals then form the source of your Analysis Services partitions.

Since the parallelism of the Process Data phase is limited by the amount of partitions you have, you should split the distinct count measure into as many equal-sized nonoverlapping intervals as you have CPU cores on the Analysis Services computer.

From Analysis Services 2005 and forward it is possible to use noninteger columns for distinct count measure groups. However, for performance reasons you should avoid this. The white paper below describes how you can use hash functions to transform noninteger columns into integers for distinct count. It also provides examples of the nonoverlapping interval-partitioning strategy.

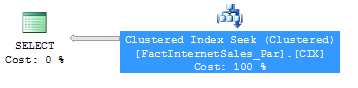
* [Analysis Services Distinct Count Optimization](http://sqlcat.com/whitepapers/archive/2008/04/17/analysis-services-distinct-count-optimization.aspx)

### Getting Relational Indexing Right

While you generally want each cube partition to touch at most one relational partition, the opposite is not true. It is perfectly viable to have to have more than one cube partition accessing the same relational partition. As an example, a relational source that is partitioned by year with a cube that is partitioned by month can still provide optimal processing performance.

If you do not have a 1-1 relationship between relational and cube partitions, you generally want an index to support the fact processing query. The best choice of index for this purpose is a clustered index; if your load strategy allows you to maintain such an index, this is what you should aim for.

When a processing query is supported by an index the plan should look like this.



**Figure 24 Index correctly supporting processing**

For more information about optimizing queries, see:

* [Top 10 SQL Server 2005 Performance Issues for Data Warehouse and Reporting Applications](http://sqlcat.com/top10lists/archive/2007/11/21/top-10-sql-server-2005-performance-issues-for-data-warehouse-and-reporting-applications.aspx)
* Itzik Ben-Gan and Lubor Kollar, *Inside Microsoft SQL Server 2005: T-SQL Querying*

#### Special Case: Distinct Count

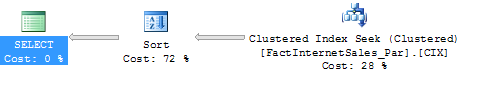
As with partitioning, distinct count is again a special case for indexing.

The distinct count processing queries will have an ORDER BY clause added to them by Analysis Services. For example, if you create a distinct count on **CustomerPONumber** in **FactInternetSales,** you would get this query while processing:

SELECT … FROM FactInternetSales

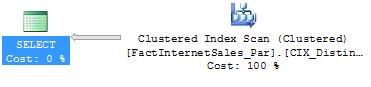
ORDER BY [CustomerPONumber]

If your partition contains a large amount of rows, ordering the data can take a long time. Without supporting indexes, the query plan will look something like this.



**Figure 25 Relational sorting caused by distinct count**

Notice the long time spent on the Sort operation? By creating a clustered index sorted on the distinct count column (in this case **CustomerPONumber**), you can eliminate this sort operation and get a query plan that looks like this.



**Figure 26 Distinct count query supported by a good index**

Of course, this index will have to be maintained. But having it in place will speed up the processing queries.

### Using Index FILLFACTOR = 100 and Data Compression

If page splitting occurs in an index, the pages of the index may end up less than 100% full. The effect is that SQL Server will be reading more database pages than necessary when scanning the index.

You can check for index pages are not full by querying the SQL Server DMV **sys.dm\_db\_index\_physical\_stats**. If the column **avg\_page\_space\_used\_in\_percent** is significantly lower than 100%, a FILLFACTOR 100 rebuild of the index may be in order. It is not always possible to rebuild the index like this, but this trick has the ability to reduce I/O. For stale data, rebuilding the indexes on the table is often a good idea before you mark the data as read-only.

In SQL Server 2008 you have the option of using either Row or Page compression to further reduce the amount of I/O required by the relational database to serve the fact process query. Compression has a CPU overhead, but reduction in I/O operations is often worth it.

## Eliminating Database Locking Overhead

When SQL Server scans an index or table, page locks are acquired as the rows are being read. This ensures that many users can access the table concurrently. However, for data warehouse workloads, this page level locking is not always the optimal strategy – especially when large data retrieval queries like fact processing access the data.

By measuring the **Perfmon** counter **MSSQL:Locks – Lock Requests / Sec** and looking for **LCK** events in **sys.dm\_os\_wait\_stats**, you can see how much locking overhead you are incurring during processing.

To eliminate this locking overhead, you have three options:

* Option 1: Set the relational database in Read Only mode before processing.
* Option 2: Build the fact indexes with ALLOW\_PAGE\_LOCKS = OFF and ALLOW\_ROW\_LOCKS = OFF.
* Option 3: Process through a view, specifying the WITH (NOLOCK) or WITH (TABLOCK)query hint.

**Option 1** may not always fit your scenario, since setting the database to read-only mode requires exclusive access to the database. However, it is a quick and easy way to completely remove any lock waits you may have.

**Option 2** is often a good strategy for data warehouses. Because SQL Server Read locks (S-locks) are compatible with other S-locks, two readers can access the same table twice, without requiring the fine granularity of page and rows locking. If insert operations are only done during batch time, relying solely on table locks may be a viable option. To disable row/page locking on a table and index rebuild ALL like this.

ALTER INDEX ALL ON FactInternetSales REBUILD

WITH (ALLOW\_PAGE\_LOCKS = OFF, ALLOW\_ROW\_LOCKS = OFF)

**Option 3** is a very useful technique. Processing through a view provides you with an extra layer of abstraction on top of the database –a good design strategy. In the view definition you can add a NOLOCK or TABLOCK hint to remove database locking overhead during processing. This has the advantage of making your locking elimination independent of how indexes are built and managed.

CREATE VIEW vFactInternetSales

AS

SELECT [ProductKey], [OrderDateKey], [DueDateKey]

,[ShipDateKey], [CustomerKey], [PromotionKey]

,[CurrencyKey], [SalesTerritoryKey], [SalesOrderNumber]

,[SalesOrderLineNumber], [RevisionNumber], [OrderQuantity]

,[UnitPrice], [ExtendedAmount], [UnitPriceDiscountPct]

,[DiscountAmount], [ProductStandardCost], [TotalProductCost]

,[SalesAmount], [TaxAmt], [Freight]

,[CarrierTrackingNumber] ,[CustomerPONumber]

FROM [dbo].[FactInternetSales] WITH (NOLOCK)

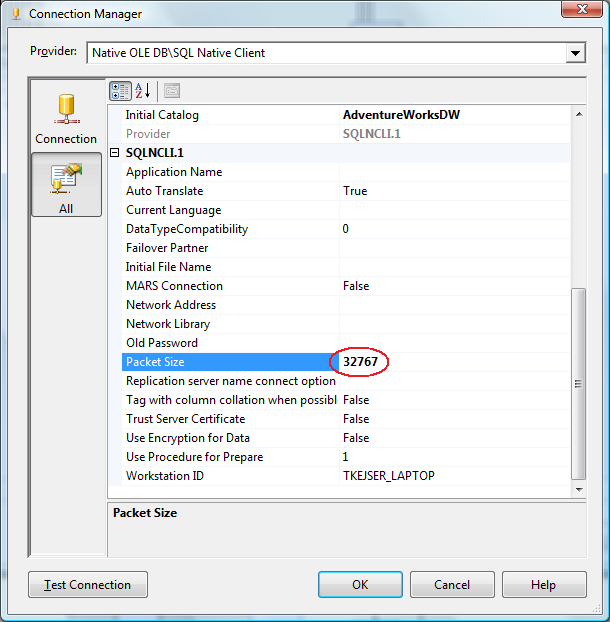
If you use the **NOLOCK** hint, beware of the dirty reads that can occur. For more information about locking behaviors, see [SET TRANSACTION ISOLATION LEVEL](http://technet.microsoft.com/en-us/library/ms173763.aspx) in SQL Server Books Online.

## Optimizing Network Throughput

During **ProcessData**, rows must be transferred between SQL Server and Analysis Services. If these two services are installed on different computers, TCP/IP network traffic occurs. You should make sure all your network components are configured to support the throughput you need. If your Ethernet throughput is consistently close to 80% of your maximum capacity, adding more network capacity will typically speed up **ProcessData**. Also, if your network is becoming a bottleneck, you will see waits for **ASYNC\_NETWORK\_IO** in SQL Server.

In addition to creating a high-speed network, there are some additional configurations you can change to further speed up network traffic.

Under the properties of your data source, increasing the network packet size for SQL Server will minimize the protocol overhead require to build many, small packages. The default value for SQL Server 2008 is 4096. With a data warehouse load, a packet size of 32K (in SQL Server, this means assigning the value 32767) can benefit processing. Instead of changing the value of the SQL Server, override it in your data source.



**Figure 27 Tuning network packet size**

You should be aware that the overhead of transporting data over the TCP/IP network is significant. Recall that Analysis Services, if it is installed on the same machine as SQL Server, is capable of using Shared Memory connections. Shared Memory connections incur minimum overhead during data exchange between SQL Server and Analysis Services. Depending on your processing workload, you may therefore be able to speed up processing by consolidating SQL server and Analysis Services to the same computer.

You can check if your connection is running shared memory by executing the following SELECT statement.

SELECT session\_id, net\_transport, net\_packet\_size

FROM sys.dm\_exec\_connections

The **net\_transport** for the Analysis Services SPID should show: **Shared memory**.

For more information about shared memory connections, see:

* [Creating a Valid Connection String Using Shared Memory Protocol](http://msdn.microsoft.com/en-us/library/ms187662.aspx)

## Improving the I/O Subsystem

If you have fully tuned the relational source system and eliminated network bottlenecks, it is time to look at the I/O subsystem.

From SQL Server’s perspective, you can measure the I/O latency from **sys.dm\_os\_wait\_stats**.If youconsistently see high waiting for **PAGELATCH\_IO**, you can benefit from a faster I/O subsystem for SQL Server.

If you have placed your Analysis Services files on a separate drive letter or mount point (which we recommend), you can use the **Logical Disk** **perfmon** counter to measure I/O wait times. If your wait times are consistently over 0.015 seconds, you can also benefit from faster I/O on the Analysis Services drives.

Techniques for tuning I/O subsystems are beyond the scope of this document. For more information, see the following technical notes and white papers:

* [Storage Top 10 Best Practices](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/storage-top-10.mspx)
* [SQL Server 2000 I/O Basics](http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/sqlIObasics.mspx)
* [Predeployment I/O Best Practices](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/pdpliobp.mspx)

## Increasing Concurrency by Adding More Partitions

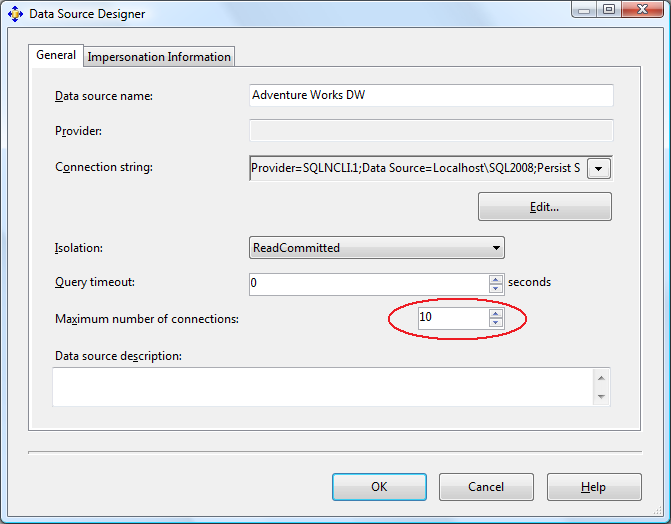
At this point of the tuning you are now bound only by the amount of CPU power you have and the ability to issue high-concurrency operations. It is time to have a look at the **Processor:Total** counter from the baseline trace. If this counter is not 100%, you are not taking full advantage of your CPU power. As you continue the tuning, keep comparing the baselines to measure improvement, and watch out for bottlenecks to appear again as you push more data through the system.

Using multiple partitions can enhance processing performance. Partitions allow you to work on many, smaller parts of the fact table in parallel. Since a single connection to SQL Server can only transfer a limited amount of rows per second, adding more partitions, and hence, more connections, can increase throughput. How many partitions you can process in parallel depends on your CPU and machine architecture. As a rule of thumb, keep increasing parallelism until you no longer see an increase in **MSOLAP:Processing – Rows read/Sec**. You can measure the amount of concurrent partitions you are processing by looking at the perfmon counter **MSOLAP: Proc Aggregations - Current Partitions**.

Being able to process multiple partitions in parallel is useful in a variety of scenarios; however, there are a few guidelines that you must follow. Keep in mind that whenever you process a measure group that has no processed partitions, Analysis Services must initialize the cube structure for that measure group. To do this, it takes an exclusive lock that prevents parallel processing of partitions. You should eliminate this lock before you start the full parallel process on the system. To remove the initialization lock, ensure that you have at least one processed partition per measure group before you begin the parallel operation. If you do not have a processed partition, you can perform a **ProcessStructure** on the cube to build its initial structure and then proceed to process measure group partitions in parallel. You will not encounter this limitation if you process partitions in the same client session and use the **MaxParallel** XMLA element to control the level of parallelism.

## Adjusting Maximum Number of Connections

When you increase parallelism of the processing above 10 concurrent partitions, you will need to adjust the maximum number of connections that Analysis Services keeps open on the database. This number can be changed in the properties of the data source.



**Figure 28 Adding more database connections**

Set this number to at least the number of partitions you want to process in parallel.

## Adjusting ThreadPool and CoordinatorExecutionMode

These server-wide properties increase the number of threads that can be used to support parallel processing operations.

**ThreadPool\Process\MaxThreads** determines the maximum number of available threads to Analysis Services during processing. On large, multiple-CPU machines, the default value of this setting may be too low to take advantage of all CPU cores. However, as you increase this counter, bear in mind that increased parallelism of processing also has an effect on queries running at the system. As you dedicate more CPU power and threads to processing, less CPU will be use for query responses. Of course, if you are processing the cubes during a batch window, this may not be an issue.

To optimize these settings for the **ProcessData** phase, check your **perfmon** counter on the object **MSOLAP: Threads** and use the table below for guidance.

| Situation | Action |
| --- | --- |
| **Processing pool job queue length** > 0 and  **Processing pool idle threads** = 0  for longer periods during processing. | Increase **Threadpool\Process\MaxThreads** and retest. |
| Both **Processing pool job queue length** > 0and **Processing pool idle threads** > 0 at same time during processing. | Decrease **CoordinatorExecutionMode** and retest. |

You can use the **Processor –% Processor Time – Total** counter as a rough indicator of how much you should change these settings. You are aiming to get as close to 100% CPU utilization as possible. For example, if your CPU load is 50% you can double **Threadpool\Process\MaxThreads** to see if this also doubles your CPU usage.

For more information about adjusting thread pools, see the following white paper:

* [SQL Server 2005 Analysis Services (SSAS) Server Properties](http://www.microsoft.com/technet/prodtechnol/sql/2005/ssasproperties.mspx)

## Adjusting BufferMemoryLimit

**OLAP\Process\BufferMemoryLimit** determines the size of the fact data buffers used during partition processing. While the default value of the **OLAP\Process\BufferMemoryLimit** is sufficient for most deployments, you may find it useful to alter the property in the following scenario.

If the granularity of your measure group is more summarized than the relational source fact table, you may want to consider increasing the size of the buffers to facilitate data grouping. For example, if the source data has a granularity of day and the measure group has a granularity of month; Analysis Services must group the daily data by month before writing to disk. This grouping occurs within a single buffer and it is flushed to disk after it is full. By increasing the size of the buffer, you decrease the number of times that the buffers are swapped to disk. Because this allows higher compression ratio, the size of the fact data on disk is decreased, which provides higher performance. However, be aware that high values for the **BufferMemoryLimit** will use more memory. If memory runs out, parallelism is decreased.

## Tuning the Process Index Phase

During the **ProcessIndex** phase the aggregations in the cube are built. At this point, no more activity happens in the Relational Engine, and if Analysis Services and SQL Server are sharing the same box, you can dedicate all your CPU cores to Analysis Services.

The key figure you optimize during **ProcessIndex** is the performance counter **MSOLAP:Proc Aggregations – Row created/Sec.** As the counter increases, the **ProcessIndex** time decreases. You can use this counter to check if your tuning efforts improve the speed.

### Avoid Spilling Temporary Data to Disk

During processing, the aggregation buffer determines the amount of memory that is available to build aggregations for a given partition. If the aggregation buffer is too small, Analysis Services supplements the aggregation buffer with temporary files. Temporary files are created in the **TempDir** folder when memory is filled and data is sorted and written to disk. When all necessary files are created, they are merged together to the final destination. Using temporary files can potentially result in some performance degradation during processing. To monitor any temporary files used during processing, review **MSOLAP:Proc Aggregations\Temp file bytes written/sec**.

In addition, when processing multiple partitions in parallel or processing an entire cube in a single transaction, you must ensure that the total memory required does not exceed thevalue of the **Memory\TotalMemoryLimit** setting. If Analysis Services reaches the **Memory\TotalMemoryLimit** during processing, it does not allow the aggregation buffer to grow and may cause temporary files to be used during aggregation processing. Furthermore, if you have insufficient virtual address space for these simultaneous operations, you may receive out-of-memory errors. If you have insufficient physical memory, memory paging will occur. If processing in parallel and you have limited resources, consider doing less in parallel.

Under the default configuration, Analysis Services will throw an out-of-memory exception if you try to request too much memory during processing. It is possible to disable this error by setting the **MemoryLimitErrorEnabled** to **false** in the server properties. However, this may cause disk spill and slow down the processing operation.

If there is no way you can avoid spilling data to disk, you should at least make sure the **TempDir** folder and Page file is a fast I/O system.

### Eliminate I/O Bottlenecks

During **ProcessIndex** the disk activity is generally lower than during **ProcessData**. If you have enough I/O to not bottleneck on **ProcessData**, chances are that the I/O speed will be sufficient for **ProcessIndex** too. However, you should still monitor the I/O using the guidelines from Improving the I/O Subsystem.

### Add Partitions to Increase Parallelism

As was the case with **ProcessData**, processing more partitions in parallel can speed up **ProcessIndex**. The same tuning strategy applies: Keep increasing partition count until you no longer see an increase in processing speed.

### Tune Threads and AggregationMemorySettings

During **ProcessIndex**, Analysis Services will scan and aggregate the partitions created during **ProcessData**. There are two ways to perform this operation in parallel:

* Use several threads to concurrently scan and aggregate the segments of one partition at a time.
* Scan and aggregate several partitions at the same time with a lower number of threads.

Both techniques can be used at the same time, to a lesser or greater degree. Using server properties, you can control how this is executed. Also, both settings are limited by the thread settings on the server, as described in Adjusting ThreadPool and CoordinatorExecutionMode.

If you are adding more partitions to increase parallelism you may need to change the **AggregationMemory** settings. If your design does not allow you to add more partitions, you have the option of changing **CoordinatorBuildMaxThreads** to further increase parallelism.

When you measure CPU utilization with the counter **Processor –% Processor Time – Total** that is less than 100%, and assuming you have no I/O bottlenecks, there is a good chance that you can increase the speed of **ProcessIndex** phase further by using the techniques in this section.

#### Adjusting Thread Settings

Just as was the case under **ProcessData** you may have to adjust your thread pool setting to achieve optimal performance. Use the guidelines from Adjusting ThreadPool and CoordinatorExecutionMode.

#### Adjusting the AggregationMemoryMin Setting

Under the server properties you will find the settings:

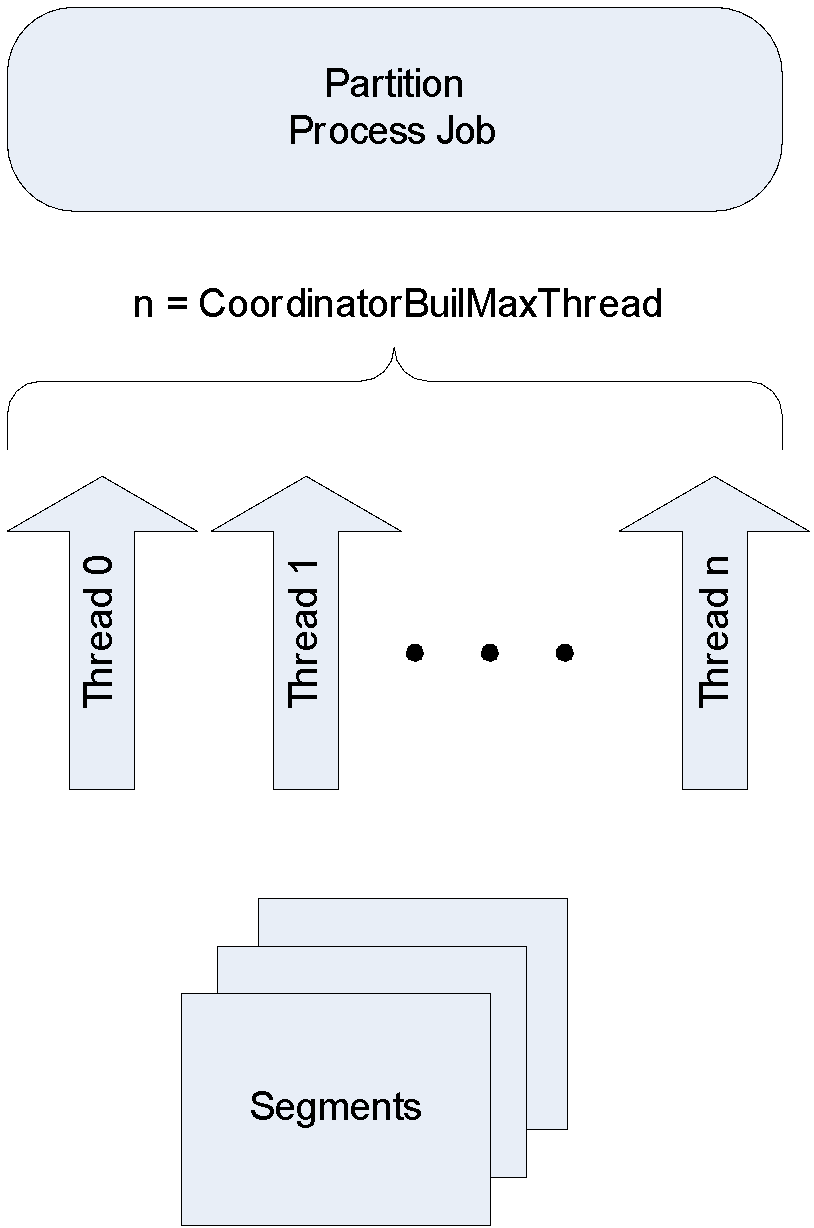
* **OLAP\Process\AggregationMemoryLimitMin**
* **OLAP\Process\AggregationMemoryLimitMax**

These settings, expressed as a percentage of the Analysis Services memory, determine how much memory is allocated for the creation of aggregations in each partition. When Analysis Services starts partition processing, parallelism is throttled based on the **AggregationMemoryMin** setting. For example, if you start five concurrent partition processing jobs with **AggregationMemoryMin** = 10, an estimated 50% of memory will be allocated for the processing. If memory runs out, new partition processing jobs will block while they way for memory to become available. If you process many partitions in parallel, lowering the value of **AggregationMemoryLimitMin** can increase **ProcessIndex** speed. By limiting the minimum amount of memory allocated per partition, you can drive a higher degree of parallelism in the process index phase.

Like the other Analysis Services counters, if this setting has a value greater than 100 it is interpreted as a fixed amount of kilobytes. For machines with large amounts of memory, using an absolute kilobyte value may provide a better control of memory than using a percentage value.

#### Adjusting CoordinatorBuildMaxThreads

When scanning a single partition, the amount of threads used to scan extents is limited by the **CoordinatorBuildMaxThreads** setting. The setting determines the maximum number of threads allocated per partition processing job. It behaves in the same way as **CoordinatorExecutionMode** (refer to the section on Job Architecture.) If this setting has a negative value, its absolute value is multiplied by the number of cores in the machine to determine the maximum number of threads that can run. If the setting has a positive value, it is the absolute number of threads that can be run. Keep in mind that you will still be limited by the number of threads in the process threadpool when processing, so increasing this value may mean increasing the process threadpool too.



**Figure 29 CoordinatorBuildMaxThreads**

If you are not able to drive high parallelism by using more partitions, you can change the **CoordinatorBuildMaxThreads** value. Increasing this allows you to use more threads per partition.

Note that you may also need to adjust the **AggregationMemoryMin** settings too, to get optimal results.

# Tuning Server Resources

While you can tune your application, there are times when you simply need to add more hardware of tune the server itself. This section describes server wide settings you can apply to increase performance.

## Using PreAllocate

The **PreAllocate** setting found in msmdsrv.ini can be used to reserve physical memory for Analysis Services. For installations where Analysis Services coexists with other services on the same machine, setting **PreAllocate** can provide a more stable memory configuration.

Note that if the service account used to run Analysis Services also has the **Lock pages in Memory** privilege, **PreAllocate** will cause Analysis Services to use large memory pages. **Lock pages in Memory** is set using gpedit.msc. Bear in mind that large memory pages cannot be swapped out to the page file. While this can be an advantage from a performance perspective, a high number of allocated large pages can cause the system to become unresponsive.

You should generally leave around 20% of total system memory for the operation system when using **PreAllocate** with large pages.

**Important:** PreAllocate has the largest impact on the Windows Server® 2003 operating system. With the introduction of Windows Server 2008, memory management has been much improved. We have been testing this setting on Windows 2008 Server, but have not measured any benefits of using **PreAllocate** on this platform. Considering the drawbacks of **PreAllocate**, there is probably very little benefit of this setting under Windows Server 2008.

To learn more about the effects of **PreAllocate**, see the following technical note:

* [Running Microsoft SQL Server 2008 Analysis Services on Windows Server 2008 vs. Windows Server 2003 and Memory Preallocation: Lessons Learned](http://sqlcat.com/technicalnotes/archive/2008/07/16/running-microsoft-sql-server-2008-analysis-services-on-windows-server-2008-vs-windows-server-2003-and-memory-preallocation-lessons-learned.aspx)

## Disable Flight Recorder

Flight Recorder provides a mechanism to record Analysis Services server activity into a short-term log. Flight Recorder provides a great deal of benefit when you are trying to troubleshoot specific querying and processing problems; however, it introduces a certain amount of I/O overheard. If you are in a production environment and you do not require Flight Recorder capabilities, you can disable its logging and remove the I/O overhead. The server property that controls whether Flight Recorder is enabled is the **Log\Flight Recorder\Enabled** property. By default, this property is set to **true**.

## Monitoring and Adjusting Server Memory

Generally, the more memory you have the better. If the data files can reside in the operating system cache, storage engine performance is very forgiving. If the formula engine can cache its results, cell values are reused rather than recomputed. During processing, not spilling results to disk also improves performance. However, be aware that Analysis Services will not use AWE memory on 32-bit systems. If your cube requires a high amount of memory, we highly recommend 64-bit hardware.

Key memory settings are **Memory\TotalMemoryLimit** and **Memory\LowMemoryLimit** and are expressed as a percentage of available memory. You can monitor memory from Task Manager or from the following the performance counters:

* MSAS2008:Memory\Memory Usage Kb
* MSAS2008:Memory\Memory Limit Low Kb
* MSAS2008:Memory\Memory Limit High Kb

Unless **PreAllocate** is used, Analysis Services gives up memory when not under load – other applications (such as the SQL Server engine) may consume freed memory and not give it up. So, it is important to configure not only Analysis Services properly but also other applications on the machine.

Before deciding that more memory is required, take the steps outlined in the querying and processing sections to optimize cube performance.

# Conclusion

This document provides the means to diagnose and address SQL Server 2008 Analysis Services processing and query performance issues. For more information, see:

<http://sqlcat.com/>: SQL Customer Advisory Team

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

If you have any suggestions or comments, please do not hesitate to contact the authors. You can reach Richard Tkachuk at [richtk@Microsoft.com](mailto:richtk@Microsoft.com?subject=Performance%20White%20Paper) or Thomas Kejser at [tjkejser@Microsoft.com](mailto:tjkeyser@Microsoft.com?subject=Performance%20White%20Paper).

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20SQL%20Server%20Analysis%20Services%202008%20Performance%20Guide).